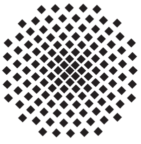

H L R I S

Institut für
Höchstleistungsrechnen

FORSCHUNGS- UND ENTWICKLUNGSBERICHTE

VERALLGEMEINERTE GLOBAL ADDRESS SPACE NETZWERK-INFRASTRUKTUREN FÜR GROSSE MULTISKALEN-SIMULATIONEN MIT ERHEBLICHEN DATENMENGEN

Thomas Großmann

ISSN 0941 - 4665 November 2014 HLRS-09

Höchstleistungsrechenzentrum
Universität Stuttgart
Prof. Dr.-Ing. Dr. h.c. Dr. h.c. M. Resch
Nobelstrasse 19 - 70569 Stuttgart
Institut für Höchstleistungsrechnen

VERALLGEMEINERTE GLOBAL ADDRESS SPACE NETZWERK-INFRASTRUKTUREN FÜR GROSSE MULTISKALEN-SIMULATIONEN MIT ERHEBLICHEN DATENMENGEN

von der Fakultät Energie-, Verfahrens- und Biotechnik
der Universität Stuttgart zur Erlangung der Würde eines
Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung

vorgelegt von

Thomas Großmann
aus Rehlingen-Siersburg / Deutschland

Hauptberichter:	Prof. Dr.- Ing. Dr. h.c. Dr. h.c. Michael Resch
MItberichter:	Prof. Dr.-Ing. Stefan Wesner
Tag der Einreichung:	18. Dezember 2013
Tag der mündlichen Prüfung:	16. Oktober 2014

D93

ISSN 0941 - 4665 November 2014 HLRS-9

Alle Rechte vorbehalten.

©2014 by Thomas Großmann

Höchstleistungsrechenzentrum Stuttgart (HLRS)
Universität Stuttgart
Nobelstraße 19
D-70569 Stuttgart

Danksagungen

Hiermit möchte ich Erich Focht (NEC) und Danny Sternkopf (ehemalig NEC) danken, welche meine Arbeit erst ermöglichten. Weiterhin möchte ich auch Professor Resch vom HLRS danken, welcher meiner Arbeit betreut hat. Ich möchte mich bei allen bedanken, die mich unterstützt haben.

Inhaltsverzeichnis

Glossary ix

1 Einleitung 6
1.1 Wissenschaftliche Anwendungen bewegen immer höhere Datenmengen . . 6
1.2 Problemstellung 8
1.3 Übersicht über die Kapitel 9

2 Stand der Technik 11
2.1 IO-Forwarding in anderen Frameworks 11
2.1.1 IOFSL 11
2.1.2 IBM Blue Gene/P 12
2.1.3 Cray Data Virtualization System 12
2.1.4 Zusammenfassung 13
2.2 Beispiele von IO in Projekten 13
2.2.1 IBM Blue Gene/P 13
2.2.2 Simulation mit Knochenmaterial 14
2.2.3 Science Experimental Grid Laboratory 15
2.2.4 Automatische Formoptimierung von hydraulischen Maschinen . . 18
2.2.5 Distributed European Infrastructrure for Supercomputing Applications 19
2.2.6 Große Multiskalen-Simulation 20

3 Realisierung IO-Forwarding mit IOFWD 21
3.1 IO-Forwarding 21
3.2 GASNet 22
3.2.1 Aufbau und Relation zu IOFWD 22
3.2.2 Funktion 22
3.2.3 Request und Reply 23
3.2.4 Heterogenität 24
3.3 Kategorisierung der Aufrufe 26
3.3.1 Öffnen und Schliessen 26
3.3.2 Schreiben und Lesen 26
3.3.3 Meta-Daten und Ordner 27
3.3.4 Datei-Verknüpfung 28
3.3.5 Verschiedenes 28
3.4 Grundlegende Informationsstruktur 29

3.5 Endianness . . . 30
3.6 Initialisierung . . . 31
3.7 Benutzung von IO-Funktionen . . . 37
3.8 Threaded Umgebung . . . 46
3.9 Sammlung von IO-Aufrufen . . . 49
3.10 Abstrahierung . . . 50

4 Einbindung von IOFWD in eine Anwendung 52
4.1 IOFWD und Anwendung verbinden . . . 52
4.1.1 Anwendung wird erstellt . . . 52
4.1.2 Erstellte Anwendung . . . 57
4.2 Zugriff auf Dateisysteme ohne Dateisystem-Client . . . 63

5 Native Anwendung 65
5.1 Erstellung der Anwendung . . . 65
5.2 Test-Aufbau . . . 65
5.3 Durchführung der Tests . . . 65

6 PGAS-Run-time mit IOFWD 71
6.1 Was ist PGAS ? . . . 71
6.2 PGAS Sprache Unified Parallel C . . . 72
6.3 PGAS Sprache UPC mit IOFWD und GASNet . . . 73
6.3.1 Einführung . . . 73
6.3.2 Compile Wrapper . . . 75
6.3.3 Starten und Zuordnen von Prozessen . . . 76
6.3.4 Handler . . . 79
6.3.5 Barriers . . . 80
6.3.6 Benchmark . . . 81

7 Simulations-Anwendung mit IOFWD 85
7.1 Beschreibung der Simulation . . . 85
7.2 Erstellen der Simulations-Applikation . . . 87
7.3 Durchführung der Simulation mit IOFWD . . . 87

8 Schlussfolgerungen 97

9 Ausblick 100

Literaturverzeichnis 104

A IO-Systemaufrufe 108

B PGAS Unified Parallel C 110

C PGAS Co-array Fortran 113

D Unterschiedliche Art von Übertragung in GAS-Netzwerken 114
D.1 UDP 114
D.1.1 Aufbau 114
D.1.2 Übertragung 116
D.2 Infiniband 116
D.2.1 Aufbau 116
D.2.2 Übertragung 117

E Speicherhandhabung 121
E.1 Memory Mapping 121
E.2 Pinning von Speicher 122
E.3 Alignment 122

Abbildungsverzeichnis

1.1 Rechenknoten tauschen IO-Daten mit dem Speichersystem aus 6
1.2 IO unterbricht die Berechnungen und verlängert den Gesamtablauf . . . 7

2.1 Biomechanische Simulation von Knochenimplantatsystemen und dabei anfallende Datenmengen 15
2.2 Biomechanische Simulation von Knochenimplantatsystemen mit Ablauf Input und Output von Dateien 16
2.3 Transport von Daten innerhalb von SEGL 17
2.4 Workflow automatische Formoptimierung von hydraulischen Maschinen mit OpenFOAM 18
2.5 DEISA-Netzwerkinfrastruktur mit 10Gbit-Anbindung 19

3.1 Übersicht IO-Forwarding: Weiterleitung eines Schreibbefehls 21
3.2 Von IOFWD benutzte Request-Reply Sequenz 23
3.3 Links Operation put und rechts Operation get für Dateiinhalte 24
3.4 get/put Layer 25
3.5 IOFWD Basis-Struktur 29
3.6 Endianness 30
3.7 Endianness-Struktur 31
3.8 Endianness-Konvertierung 32
3.9 IOFWD Initialisierung: Übersicht 33
3.10 IOFWD Server Hauptschleife 34
3.11 IOFWD-Initialisierung: Füllen der Strukturen 35
3.12 IOFWD Initialisierung: Einschalten des IO-Forwarding 36
3.13 Spezieller Pfad für IO-Forwarding 37
3.14 File Deskriptoren Übersetzung bei open Funktion 38
3.15 File Deskriptoren Übersetzung bei close Funktion 39
3.16 Kompletter Aufruf-Pfad von open Client nach Server 40
3.17 Kompletter Aufruf-Pfad von open Server nach Client 40
3.18 Serialisierung im Client am Beispiel read 41
3.19 Deserialisierung im Server am Beispiel read 41
3.20 Aufrufkette read Client nach Server 42
3.21 Aufrufkette read Server nach Client 43
3.22 Aufrufkette write Client nach Server 44
3.23 Aufrufkette write Server nach Client 44
3.24 Bewegen von Dateiinhalten von IOFWD Server aufgrund von write . . . 45
3.25 Bewegen von Dateiinhalten von IOFWD Server aufgrund von read 46

3.26 Ein Client operiert mit read mit mehreren Servern 47
3.27 Ein Client operiert mit write mit mehreren Servern 47
3.28 IOFWD Transferpuffer mit mehrfachen Kommandopuffern 48
3.29 Ein Client mit 3 read Threads auf 3 Servern 48
3.30 Ein Client mit 3 write Threads auf 3 Servern 49
3.31 Sammeln von IO-Aufrufen 50
3.32 Abstraktionsebenen von IOFWD 51

4.1 Compile-Wrapper Preprocessing 53
4.2 Compile-Wrapper outputfile Subroutine 54
4.3 Compile-Wrapper cending Subroutine 55
4.4 Compile-Wrapper Wrap, Compile und Link Prozedur 56
4.5 Aufruf-Präferenz 60

5.1 Anwendung nbmrw mit Threaded IOFWD Client 66
5.2 Lesen ohne IOFWD über natives Infiniband Linux-zu-Linux 67
5.3 Lesen mit IOFWD über natives Infiniband Linux-zu-Linux 68
5.4 Schreiben ohne IOFWD über natives Infiniband Linux-zu-Linux 69
5.5 Schreiben mit IOFWD über Infiniband Linux-zu-Linux 69
5.6 Lesen/Schreiben von einzelnen 1-kByte-Blöcken ohne IOFWD über Infiniband Linux-zu-Linux 70
5.7 Lesen/Schreiben 1-kByte-Blöcke zusammengefasst mit IOFWD über Infiniband Linux-zu-Linux 70

6.1 Rechen- und IO-Server Prozesse auf zwei Cluster verteilt 74
6.2 Einfügen des IOFWD Transfersegments 75
6.3 Starten der Prozesse in UPC 76
6.4 Liste der Orte für die Prozesse 77
6.5 Prozessliste mit doppelten Nennungen in Reihenfolge 77
6.6 Prozessliste mit doppelten Nennungen nicht in Reihenfolge 78
6.7 Starten der unterschiedlichen Prozesse: relative Prozessnummern 79
6.8 Starten der unterschiedlichen Prozesse: absolute Prozessnummern 80
6.9 Starten der unterschiedlichen Prozesse: mod-Rechnung für Verteilung auf IO-Server 81
6.10 UPC-Initialisierung: Starten der unterschiedlichen Prozesse 82
6.11 UPC-Initialisierung: Aufteilung der Barriers 82
6.12 Shared Array-Zuweisung auf Knoten 0: put wird ausgelöst 83
6.13 Shared Array-Zuweisung auf Knoten 0: get wird ausgelöst 83
6.14 GASNet put-Operation von Thread 0 nach Thread 1 Schreiben von Thread 0 nach Thread 1 84
6.15 GASNet get-Operation von Thread 1 nach Thread 0 Schreiben von Thread 1 nach Thread 0 84

7.1 Allgemeine Konfiguration eine aero-akustischen Simulation. Um die Tragfläche befindet sich ein unstrukturiertes Gitter auf der der physikalische Fluß berechnet werden muß (a). Diese Domain ist eingebettet in einer Fernfeld Domain mit einem strukturierten Gitter, in der die Wellenausbreitung berechnet wird (b). 85
7.2 Simulation mit getrennten File Systemen 88
7.3 Datentransfer: lokales SX GFS Dateisystem 88
7.4 Simulation mit einem File System (Lustre) 89
7.5 6 SX-Knoten verbinden sich zu IOFWD Server 90
7.6 Zeitpunkte und transferierte Dateien für 2 SX-Knoten 91
7.7 Transferierte Mengen für 2 SX-Knoten im Detail 92
7.8 Transferierte Gesamt-Mengen für 2-6 SX-Knoten 93
7.9 Datentransfer: Schreiben 2 SX-Knoten zu IOFWD Linux Server 94
7.10 Datentransfer: Schreiben 3 SX-Knoten zu IOFWD Linux Server 94
7.11 Datentransfer: Schreiben 4 SX-Knoten zu IOFWD Linux Server 95
7.12 Datentransfer: Schreiben 5 SX-Knoten zu IOFWD Linux Server 95
7.13 Datentransfer: Schreiben 6 SX-Knoten zu IOFWD Linux Server 96
7.14 Datentransfer: Schreiben 2-6 SX-Knoten zu IOFWD Linux Server, gemittelter Restart-Durchschnitt pro Knoten 96

D.1 Kommunikation der Master- und Worker-Knoten 115

Tabellenverzeichnis

A.1 Unterstützte IO-Systemaufrufe mit Anzahl Argumente 109

Glossary

API Application Programming Interface.

HCA Host Channel Adapter.

PGAS Partitioned Global Address Space.

SPMD Single-Program Multiple-Data.

TCA Target Channel Adapter.

TCP Transport Control Protocol.

UDP User Datagram Protocol.

UPC Unified Parallel C.

Zusammenfassung

Diese Arbeit handelt von dem Software Framework IOFWD. IOFWD handhabt dateibezogenen IO-Verkehr zwischen Rechenprozessen und IO-Server-Prozessen. Ein Global Address Space Framework namens GASNet unterstützt IOFWD bei der Aufgabe von Umleitung (IO-Forwarding) von dateibezogenen Daten auch zwischen unterschiedlichen Clustern.
Eine Einleitung zu dem generellen Problem von IO in wissenschaftlichen Anwendungen und den Grund von IO-Optimierungwird gegeben. Der Optimierung von IO wurde bisher nicht diesselbe Sorgfalt gewidmet wie den nicht IO-Teilen der Applikation. IO-Performance wird mehr und mehr kritisch, weil die CPU-bezogene Hardware mehr und mehr Performance Overhead gewinnt. Das Heilmittel, das hier vorgeschlagen wird, ist die Optimierung und Redirektion von IO-Verkehr auf dem Software Level. Die IO Optimierungs-Software IOFWD soll auch einfach anzuwenden und in existierende Applikationen zu integrieren sein.
Ein von GASNet abgeleitetes Konzept wird entworfen, das auch in der Lage ist, mit anderen Frameworks zu arbeiten. Zuerst wird eine Übersicht über den gegenwärtigen Stand der Technologie gegeben. IO-Forwarding in anderen Frameworks wie IOFSL und die Konzepte von unterschiedlichen Frameworks werden gezeigt. Gefolgt von einer Serie von Beispielen meist von der Universität Stuttgart. Die Projekte wurden analysiert auf ihr IO Verhalten und den Mengen von Daten, die während der Operation der spezifischen Projekt-Applikationen bewegt wurden.
Dann wird IOFWD gezeigt in seiner Struktur, Funktion und Optimierungsstrategien. Zuerst wird das Konzept von IO-Forwarding erklärt. Dann wird GASNet, auf das IOFWD aufbaut, in Relation zu IOFWD und dessen Aufbau gezeigt. Die unterstützten IO-Funktionsaufrufe in IOFWD werden aufgelistet und innerhalb einer kategorischen Gliederung erklärt. Danach wird die Art, wie IOFWD arbeitet, im Detail erklärt. Das beinhaltet Themen wie Endianness Konversion und Handhabung, Benutzung von IO-Funktionen z.B. die Art wie IOFWD zwischen lokalen und IOFWD bezogenen Verkehr unterscheidet. Threads werden innerhalb IOFWD insofern eingeführt, dass ein IOFWD Client auf multiplen IOFWD Servern arbeitet. Der Load der IOFWD Clients wird gleichmäßig und simultan auf multiple IOFWD Server verteilt. Die Sammlung von IO-Aufrufen und deren Inhalt ist ebenfalls ein Thema in diesem Kapitel. Schließlich wird die Abstraktionsschicht innerhalb IOFWD, die verschiedene Frameworks wie GASNet und BMI integriert, gezeigt.
Weiterhin wird die Verbindung von einer Anwender-Applikation mit IOFWD diskutiert. Hier werden Fälle unterschieden angefangen von dem Source Code einer Applikation bis zur finalen Ausführbaren, genauso wie der statische und der dynamische Fall einer ausführbaren Datei. Besonders der Fall, wenn die ausführbare Datei statisch gelinkt ist, wird

in einem Abschnitt, der sich mit Instrumentation beschäftigt, gesprochen. In dieser Sektion wird gezeigt, wie Instrumentation genutzt werden kann, um IOFWD-Funktionalität in eine statisch ausführbare Datei einzusetzen.
Eine Test-Applikation aus den IOFWD Test Applikationen, die Performance-Daten ausgibt, wird gezeigt. Die Test Applikation schließt typische IO-Kommandos ein und testet im Grunde sequentielles Lesen und Schreiben mit unterschiedlichen Größen von IO. In diesem Fall wird die IOFWD Software komplett auf einem Linux Cluster mit Infiniband Konnektivität ausgeführt. Es wird auch erklärt, wie IOFWD in diese Test Applikation integriert wird.
Die Integration von IOFWD in die PGAS Laufzeit-Umgebung von Unified Parallel C und eine Performance Evaluation werden eingeführt. Hier läuft die PGAS UPC Umgebung auf einem NEC SX Vector Cluster und die IOFWD Server Umgebung auf einem verbundenen Linux Cluster. Die IOFWD Client Umgebung ist komplett in die PGAS UPC Laufzeit Umgebung integriert and startet problemlos und einfach mit. Die Ausführungsabschnitte der Prozesse von dem IOFWD und PGAS UPC Compartment und wie die Prozesse von UPC und IOFWD innerhalb der GASNet Umgebung starten wird gezeigt und im Detail erklärt. Es wird nur eine GASNet Umgebung für alle Knoten benutzt, im Gegensatz zur herkömmlichen Anwendung von IOFWD, in der es eine GASNet Umgebung für ein Knotenpaar (Rechenknoten und IO Server Knoten) gibt. Die Integration von IOFWD in die Compile Umgebung wird erklärt an der PGAS Beispiel Applikation, die einfach Inhalte von Arrays hin- und herkopiert. Der IOFWD Compile Wrapper übersetzt Compile Kommandos in Kommandos, die IOFWD in die Applikation mit einschließen, so daß die finale Ausführbare IOFWD Funktionalität benutzt.
Die Interaktion von IOFWD mit einer multiskalen Simulation (MPI-basiert) wird gezeigt. Performance Werte folgen anschließend. Die multiskale Simulation, die IO-Datenstrukturen und deren Performance werden beschrieben. Hier ist ein NEC SX Vector Cluster und mit einem Linux Cluster verbunden. Der NEC SX Vector Cluster hat die Rechenknoten einschließlich IOFWD Client Funktionalität und der Linux Cluster bietet die IOFWD Server Funktionalität. Die unterschiedlichen Phasen von IO und die Timings von IO Output werden aufgezeigt. Eine detailierte Beschreibung von den Mengen von Daten von IO, die von der Simulation produziert werden in den verschiedenen Phasen der Simulation, werden herausgestellt. Es existiert auch ein Abschnitt, der erklärt, wie die Simulation mit IOFWD verbunden wurde.
Schließlich werden Schlußfolgerungen gezogen und weitere mögliche Arbeit wird diskutiert.

Summary

This thesis deals about the software framework IOFWD. IOFWD handles file related IO traffic between calculating processes and IO server processes. A global address space framework named GASNet supports IOFWD at the task of redirection (IO-Forwarding) of file related data also between different clusters.
An introduction is given to the general problem of IO in scientific applications and the reason for IO optimisation. Generally the reason is, that optimisation of IO has not been taken care of the same way than optimisation of non-IO parts of an application. IO performance becomes more and more critical, because the CPU-related hardware is gaining more and more performance overhead. The remedy, which is suggested here, is an optimisation and redirection of the IO-traffic at a software level. The IO optimisation software IOFWD should be also easy to use and to integrate into existing applications. A concept derived from GASNet will be developped, that is also able to work with other frameworks. Firstly an overview at the current state of technology is given. IOFWD forwarding in other frameworks like IOFSL is shown and the concepts of the different frameworks are explained. Followed by a series of examples of projects mostly from the university of stuttgart. The projects were analyzed for their IO behaviour and the amounts of data that were moved during application operation of the specific project applications.
Then IOFWD is shown in its structure, function and optimisation strategies. First the concept of IO forwarding is explained. Then GASNet, on which IOFWD is build, is shown in relation to IOFWD and its build-up. The supported IO function calls in IOFWD are listed and explained in a categorical manner. After that the way IOFWD works, is discussed in detail. That includes topics like endianness conversion and handling, use of IO functions e.g. the way IOFWD distinguishes between local and IOFWD related traffic. Threads have been introduced into IOFWD so far, that an IOFWD-client operates on multiple IOFWD-servers. The load of the IOFWD-client is distributed equally and simultaniously amongst multiple IOFWD servers. Collection of IO-calls and their content is also a topic in this chapter. Finally the abstraction layer within IOFWD in which different frameworks like GASNet and BMI are integrated is shown.
Furtermore the connection of an application with IOFWD is discussed. Here are the cases distinguished from given source code of an application and a final executable, as well as the static and dynamic case of an executable file. Especially the case, where the executable file is statically linked, is talked about in a section dealing with instrumentation. In this section it is shown, how instrumentation can be used, to insert IOFWD functionality into a static executable file.
A test application out of the IOFWD test-suite of applications, which gives out perfomance data, is shown. The test application incooperates typical IO commands and

basically tests sequential reads and writes with various sizes of IO. In this case, the IOFWD software is run completely on a Linux cluster with Infiniband connectivity. It is also explained, how IOFWD is being integrated with the test application.
The integration of IOFWD into the PGAS run time environment of Unified Parallel C and performance evaluation is introduced. Here the PGAS UPC environment runs on the NEC SX vector cluster and the IOFWD server environment runs on a connected Linux cluster. The IOFWD client environment is completely integrated into the PGAS UPC run time environment and starts up seamlessly with it. The execution sections of the processes from the IOFWD and PGAS UPC compartment and how the processes of UPC and the IOFWD processes within the GASNet environment are started up is shown and explained in detail. There is just one GASNet network used for all nodes, whereas in the non-PGAS solution there exists a GASNet network for each node pair (calculation node and io-server node). The integration of IOFWD into the compile environment is explained on behalf of this PGAS example application which simply moves contents of arrays around. The IOFWD compile wrapper translates compile commands into commands that incooperate IOFWD into an application, so that the final executable uses IOFWD functionality.
The interaction of IOFWD with a multiscale simulation (mpi-based) and performance values are following thereafter. The multiscale simulation, the IO data structure and its performance are being described. Here a NEC SX vector cluster and a linux cluster are connected. The NEC SX vector cluster has the computing nodes including IOFWD IO client functionality and the linux cluster is providing the IO-server functionality. The different phases of IO and the timings of IO output are pointed out. A detailed description of the amounts of IO which are produced by the simulation application in the different phases of the simulation is being brought forward. There is also a section how the simulation application has been created with IOFWD.
Finally conclusions are drawn and further possible work is discussed.

Vorwort

Die in dieser Arbeit beschriebene Software entspringt aus dem Teilprojekt IOFWD [1] und ist innerhalb einer Kooperation vom Höchstleistungsrechenzentrum (HLRS) der Universität Stuttgart und NEC entstanden. Es ist Teil des Projekts SX-Linux [2, 3].

1 Einleitung

1.1 Wissenschaftliche Anwendungen bewegen immer höhere Datenmengen

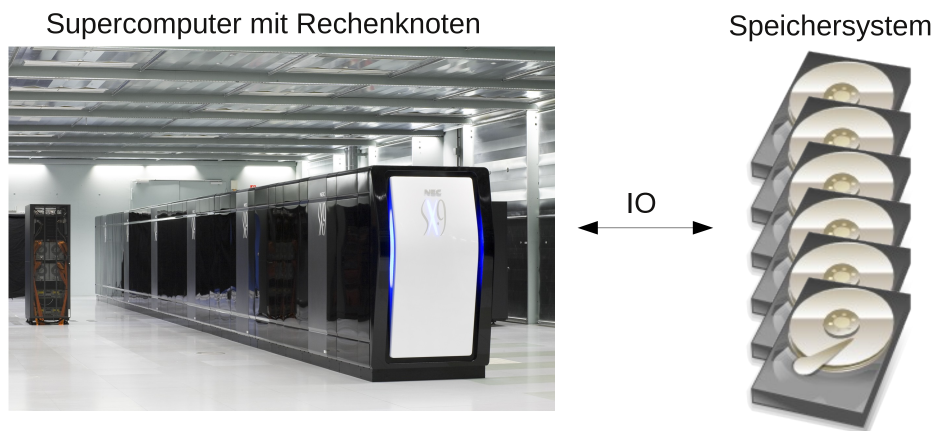

Abbildung 1.1: Rechenknoten tauschen IO-Daten mit dem Speichersystem aus

IO bedeutet Input und Output. Es werden Daten vom Rechenprozess auf dem Rechenknoten weggeschickt (Output) und Daten vom Rechenprozess auf dem Rechenknoten entgegengenommen (Input). Die Daten, die hier von Relevanz sind, sind die dateibezogenen Daten. Genauer gesagt, die Meta-Daten von Dateien und Dateiinhalte. Zu den Meta-Daten gehören z.B. Schreib- und Leserechte für Benutzer, Zeitpunkte für Zugriffe und Position des Zugriffspunktes auf die Datei. Primär wird der IO abgelegt im Dateisystem, das die Daten auf das Speichersystem transportiert d.h. der Rechenknoten hat einen Dateisystem-Client, der sich um IO-Aufgaben kümmern muß.
Während die Rechenleistung der Prozessoren in den Rechenarchitekturen nach Moore's Gesetz stetig ansteigt, steigt die Leistung des IO-Systems insbesondere der Speichersysteme (meist Festplattensysteme) nicht in entsprechendem Maße an. Die Rechenleistung wächst überproportional zur IO-Leistung von Rechensystemen. Man spricht hier auch von IO-Bandbreite pro Spitzenleistung in Teraflop. Während bei früheren Rechensysteme das Verhältnis etwa 1 GBps für ein Teraflop war, ist es bei gegenwärtigen führenden

Rechensystemen 1 GBps für 10 Teraflops. Kann die Datenmenge vom IO-System nicht in geeigneter Zeit bewältigt werden, werden die Prozessoren hiervon ausgebremst (Abbildung 1.2). Die Anwendung braucht mehr Zeit, um ihre Aufgabe zu erledigen.

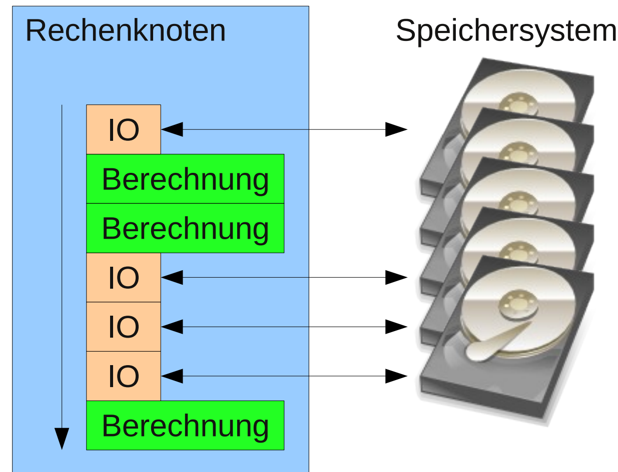

Abbildung 1.2: IO unterbricht die Berechnungen und verlängert den Gesamtablauf

Die Überlegung ist natürlich den IO in Hardware und/oder Software zu beschleunigen. Hierzu sind Punkte wichtig wie die Bestimmung des Unterbrechungspunktes im Rechenknotens für die effiziente Bearbeitung von IO. Es werden zunächst augenfällige Verbesserungen diskutiert, die für Hardware und Software von Bedeutung sind. Im weiteren wird dann auch auf die Separierung des IO aus dem Rechenprozess eingegangen, der eine sinnvolle Optimierung ermöglicht. Diese Funktionstrennung vereinfacht auch die effiziente und vor allem schnelle Behandlung des IO. Dadurch werden auch Optimierungsstrategien für den IO leichter erkennbar und durchführbar. Einige der angezeigten Aspekte werden Neues zu Tage fördern, während andere reine Fleißarbeit in der Anwendung bekannter Techniken aus anderen Gebieten sind.
Betrachten wir zunächst die Verwendung schnellerer Hardware. Dies ist bei einer Neuanschaffung möglich. Existiert eine Infrastruktur bereits, ist ein Upgrade der Hardware beschränkt und verhältnismäßig teuer, wenn dies nicht entsprechend in einer früheren Planung schon vorbereitet wurde.
Schnellere Rechenknoten bedeuten nur bedingt schnelleren IO, da Anschlusssysteme der IO-Peripherie entsprechend Leistung bringen müssen. Die Rechenknoten sind hier weniger wichtig. Daher konzentrieren wir uns zur Anschaulichmachung auf zwei Gebiete der Hardware:

- Festplattensysteme
- Netztransportsysteme

Eine positive Entwicklung am Markt der Festplattensysteme sind SSDs (Solid-State Drives) [4]. SSDs arbeiten im Gegensatz zu herkömmlichen Festplatten nicht mit beweglichen Bauteilen. Hier werden die Daten, wie beim USB-Stick, mit Hilfe von Flash-Bausteinen geschrieben und gelesen. Die SSD-Festplatte verursacht keine Geräusche und verbraucht mit ca. 2 Watt bei 128 Gbyte Größe wenig Strom. Während eine herkömmliche Festplatte eine Transferrate zwischen 80 und 100 Mbyte/s zum Lesen/Schreiben bietet, liegt diese bei SSDs bei 300-500 Mbyte/s Peak-Performance (Serial ATA-600). Noch schnellere Übertragungsraten mit schnelleren Übertragungsschnittstellen sind in Arbeit. Die SSD hat momentan noch den Nachteil, daß die verfügbare Kapazität noch relativ teuer ist und ein Vielfaches im Vergleich zur herkömmlichen Festplatte kostet. Also ist diese Lösung für die nahe Zukunft noch nicht kostengünstig realisierbar.
Die zweite Möglichkeit, schnellere Hardware zu verwenden, ist die Verwendung schnellerer Netztransportsysteme. Hierbei ist Infiniband ein Transportmittel der Wahl. Es hat den Vorteil das die Anbindung der Festplattensysteme und die Kommunikation der Rechner untereinander allesamt mit Infiniband erfolgen können. Die Optimierung des IOs der Systeme kann von einer zentralen Stelle im Cluster administriert und analysiert werden. Die gegenwärtige und zukünftige Entwicklung von Infiniband hinsichtlich neuer Geschwindigkeitsklassen sind sehr gut und machen guten Fortschritt. Die Entwicklung geht in Richtung 25 Gbit/s für einen einfachen Kanal und bis zu 300 Gbit/s für einen 12-fach gebündelten Kanal. Infiniband wird in dieser Arbeit zu einem späteren Zeitpunkt (Abschnitt 5 auf Seite 65 und D.2 auf Seite 116) noch weiter ausgeführt.
Nach der Verwendung einer möglichst schnellen Hardware kommt die Optimierung des IOs in Software. Die Software besteht grundsätzlich aus Betriebssystem, System-Bibliothek und Applikation mit Bibliotheken. Die Optimierungsmöglichkeiten sind in diesem Fall vielzählig, können aber auch komplex und fehleranfällig sein, wenn nicht die notwendige Sorgfalt beim Design der Software getroffen wurde. Es gibt viele Benutzer-Bibliotheken und Betriebssysteme und deren Varianten, für die jeweils eine gemeinsame Optimierung des IOs erfolgen kann. Eine Lösung dafür ist eine allgemeine IO-Bibliothek, die universell auf verschiedenen Systemen einsetzbar ist und trotz der relativ hohen Abstraktion IO-Daten effizient übertragen kann.

1.2 Problemstellung

Der Rechenprozess soll beschleunigt werden. Dazu soll der IO-Anteil herabgesetzt werden und die Technik des Weiterleitens von IO an spezialisierte Prozesse (IO-Server) benutzt werden (IO-Forwarding). Es soll das Global Address Space Framework GASNet verwendet werden als Transportschicht für den IO. Aus GASNet heraus soll ein IO-Konzept entwickelt werden. Basis soll das put/get Konzept von GASNet sein. Hierbei werden Inhalte mit put/get von einem zu einem anderen Prozess verschoben, wobei das Kommando die Richtung bestimmt. IO-Prinzip: put bedeutet zum Zielhost hin und get bedeutet vom Zielhost weg.
Dazu müssen folgende Teilprobleme in dem IO-Konzept gelöst werden:

- wo wird der IO abgefangen?

- wie wird der IO verarbeitet und weitergeleitet?
- wie sieht der spezialisierte IO Prozess (IO-Server) aus?
- wie kann der IO optimiert werden?

Das IO-Konzept sollte abstrahiert und unabhängig von GASNet verwendbar sein. Der Name des IO-Frameworks, das auf diesem IO-Konzept basiert, wird IOFWD sein. Es sollen Tools für IOFWD entwickelt werden, die eine leichte Integration in eine Anwendung ermöglichen. Hierbei sind zwei Situationen zu beachten:

- die Anwendung wird erstellt
- die Anwendung ist erstellt.

Auch sind die Fälle von dynamischen und statischen Anwendungen zu unterscheiden. Es soll die Performance von IOFWD unter den beiden Netzwerkprotokollen UDP und Infiniband untersucht werden. UDP soll repräsentierend sein für Internet-weite Verbindungen und Infiniband ist ein typisches Intranet-Protokoll, das auch Cluster verbindet.

1.3 Übersicht über die Kapitel

Dieser Abschnitt gibt eine Kapitelbeschreibung und gleichzeitig eine Einsicht wie der Autor in die beschriebene Software IOFWD und den weiteren Themen involviert war. Diese Beschreibung wird als notwendig erachtet, da IOFWD ein Projekt innerhalb eines Teams von zwei Entwicklern (einschließlich des Autors) war. Im folgenden Kapitel 2 auf Seite 11 wird eine Übersicht über den Stand der Technik gegeben. Hier werden vorhandene IO-Forwarding Frameworks beschrieben. Weiterhin werden Projekte und deren IO-Bedarf vorgestellt. Die Analyse der Frameworks und der Projekte wurden alleinig vom Autor erstellt. Das nächste Kapitel 3 auf Seite 21 stellt Aufbau, Funktion und Optimierungsstrategien von IOFWD vor. Sämtliche beschriebenen Funktionen wurden vom Autor grundlegend im Rahmen der Abstrahierung der Funktionen und im Rahmen der Optimierungsstrategien bearbeitet. Bei der Abstrahierung mußten die Funktionen von BMI[1] und GASNet analysiert und neu geschrieben werden. Hier kann der Autor zumindest eine wesentliche Weiterentwicklung und auch Neuentwicklungen vorweisen. Im nachfolgenden Kapitel 4 auf Seite 52 wird allgemein das Zusammenspiel von IOFWD und einer Anwendung analysiert und geeignete Tools und Vorgehensweisen gezeigt. Dabei wird unterschieden zwischen erstellter und noch zu erstellender Anwendung. Die beschriebene Software wurde alleinig vom Autor geschrieben. Kapitel 5 auf Seite 65 zeigt Performance-Daten der IOFWD Test-Anwendung nbmrw. Die Anwendung nbmrw wurde maßgeblich vom Autor verändert und angepasst. Die Infiniband-Anpassung von IOFWD an GASNet wurde alleinig vom Autor durchgeführt. Die Integration von

[1] BMI [5] wurde vom Autor auch in die Abstraktion vollständig eingebunden, umgeschrieben und getestet. Es ist jedoch nicht Gegenstand dieser Arbeit.

IOFWD in die Laufzeitumgebung von Unified Parallel C, einer PGAS-Sprache, wird in Kaptitel 6 auf Seite 71 vorgestellt. Außerdem werden die Performance-Werte einer UPC Test-Anwendung vorgestellt. Diese Anpassung wurde alleinig vom Autor durchgeführt. Kapitel 7 auf Seite 85 stellt das Zusammenspiel von IOFWD und einer Simulation (MPI-basiert) vor und zeigt Performance-Werte. Die Versuchsdurchführung wurde alleinig vom Autor geleitet. Nachfolgend zieht Kapitel 8 auf Seite 97 Schlussfolgerungen aus den vorhergehenden Kapiteln und der gemachten Arbeit. Kapitel 9 auf Seite 100 zeigt mögliche Weiterentwicklungen und weitere Ideen zu IOFWD auf.

2 Stand der Technik

2.1 IO-Forwarding in anderen Frameworks

IO-Forwarding Frameworks sind von immer steigender Relevanz. Es wird jedoch noch zu wenig darüber publiziert. Nachstehend wird eine Auswahl wichtiger Frameworks vorgestellt.

2.1.1 IOFSL

IOFSL [6] ist ein IO-Forwarding Framework, das unter anderem eine POSIX [7] IO und eine MPI-IO [8] -Schnittstelle bietet. POSIX ist ein Application Programming Interface (API), das die Schnittstelle zwischen Applikation und Betriebssystem darstellt. Es spezifiziert unter anderem auch Systemaufrufe wie open, read, write, close, die mit Dateien arbeiten. MPI-IO ist Teil von MPI-2, einer Message Passing-Bibliothek zur parallelen Programmierung. MPI-IO beinhaltet unter anderem die Möglichkeit, Dateien blockierend und nicht blockierend zu lesen und zu schreiben. In MPI werden Nachrichten zwischen MPI-Prozessen ausgetauscht. Das Lesen und Schreiben von Dateien wird auch mit dem Versenden und Empfangen von MPI-Nachrichten erreicht. Genauer wird auf die Schnittstellen

- modifizierte GNU C-Bibliothek [9]
- POSIX via SYSIO-Bibliothek [10]
- MPI-IO via ROMIO [11]

auf der IOFSL Client-Seite zugegriffen.
Die GNU C-Bibliothek ist eine C-Bibliothek-Implementierung für UNIX-artige Betriebssysteme. Diese Bibliothek definiert unter anderem Systemaufrufe zum Betriebssystem und Aufrufe wie open, read zum Bearbeiten von Dateien. Die GNU C-Bibliothek ist in C geschrieben und unterstützt POSIX-Standards. Annähernd alle bekannten und nützlichen Funktionen von jeglicher anderer C-Bibliothek sind in der GNU C-Bibliothek verfügbar. Die GNU C-Bibliothek wird vorwiegend in GNU- und Linux Kernel-basierten Systemen verwendet. Die SYSIO-Bibliothek ist eine User-Space-Implementation einer virtuellen Dateisystemabstraktion. Vereinfacht gesagt ist dies ein allgemeines Dateisystem, das in ein spezielles Dateisystem überführt und somit der Applikation die unterschiedlichen Details des jeweils spezifischen Dateisystems verhüllt. Diese Bibliothek bietet eine POSIX-Schnittstelle zur Applikation. ROMIO ist eine Software-Implementation

von MPI-IO. Darunter befindet sich die ZOIDFS IO-Schicht. ZOIDFS ist ein zustandsloses Protokoll und hat seine Wurzeln im ZOID-Projekt [12]. Anstatt File-Deskriptoren benutzt das Protokoll opake Datei-Handler um die Objekte zu beschreiben, auf denen IO-Operationen ausgeführt werden. Unter opaken Datentypen allgemein versteht man in einer Schnittstelle unvollständig definierte Datentypen. Ein Beispiel hierfür sind Datenstrukturen, die Elemente enthalten, die zunächst nicht vollständig definiert sind. Diese werden zu einem späteren Zeitpunkt definiert. Diese Technik wird z.B. auch in IOFWD verwendet, in dem netzwerk-unabhängige Parameter in einer Datenstruktur direkt definiert werden und netzwerk-abhängige Parameter, die z.B. GASNet betreffen, zu einem späteren Zeitpunkt in dieser Datenstruktur von der Netzwerkschicht definiert werden. Da das ZOIDFS-Protokoll zustandslos ist, können diese Datei-Handler frei unter den IOFSL Clients ausgetauscht werden. Das ZOIDFS API ist weiter gefasst als das POSIX API und benötigt weniger IO-Aufrufe für Dateioperationen.
IOFSL benutzt BMI als Netzwerkabstraktion. BMI wurde als Teil des parallelen PVFS-Dateisystems [13] entwickelt. Funktionsparameter der Dateioperationen werden mit XDR [14] kodiert, um eine gewisse Heterogenität zu gewährleisten. Jeder IOFSL Server kann multiple IOFSL Clients bedienen.

2.1.2 IBM Blue Gene/P

Die IBM Blue Gene/P bietet auch IO-Forwarding. Hier sind die IO-Clients indirekt über ein Collective Tree-Netzwerk zu einem IO-Server verbunden. Das Verhältnis von IO-Clients zu IO-Servern variiert von 1:8 bis 1:64. Das IO-Forwarding Framework der Blue Gene/P skaliert sehr effizient auf großen Systemen. Der proprietäre Ansatz limitiert jedoch die Nutzbarkeit auf ein System dieser Art. Weitere Möglichkeiten auf der Blue Gene/P ist ein User-Level-Dateisystem, das auf FUSE [15] basiert oder die SYSIO-Bibliothek. Die SYSIO-Bibliothek bietet POSIX-ähnlichen Dateien IO-Unterstützung. Ein Nachteil des User-Space-Dateisystems FUSE ist, das es lediglich Unterstützung für die UNIX-Variante Linux bietet. Man muß auch darauf achten, daß FUSE-Dateisysteme auch cachen. Dies kann problematisch sein mit entfernten IO-Servern.

2.1.3 Cray Data Virtualization System

Cray's DVS (Data Virtualization System) [16] bietet die Möglichkeit, IO-Forwarding zu betreiben. DVS basiert auf Portals [17], das speziell für das darunterliegende Seastar-Netzwerk optimiert wurde [18]. DVS unterstützt multiple IO-Server, mit denen es möglich ist, eine Datei über alle IO-Server aufzuteilen. Verschiedene Variablen und Optionen steuern, wie der IO von den IO-Clients auf die IO-Server aufgeteilt wird.

2.1.4 Zusammenfassung

Die vorgestellten Lösungen haben mehr oder weniger folgende Unzulänglichkeiten:

- nur auf bestimmte Hardware zugeschnitten, auf bestimmter Hardware benutzbar
- Netzwerk-Infrastruktur Framework schwer austauschbar
- closed-source
- nur auf MPI-basierte Umgebungen zugeschnitten, nicht jedoch auf PGAS-Umgebungen
- wenig offen für weiterführende Konzepte.

Diese Unzulänglichkeiten führen zu der Idee ein neues Framework für IO-Forwarding zu entwerfen und zu implementieren. Zunächst wird jedoch an einigen Projekten deren IO-Charakteristik untersucht, um ein Gefühl dafür zu bekommen, welche Anforderungen an ein Framework hier zu Tage treten.

2.2 Beispiele von IO in Projekten

Um sich ein Bild davon zu machen, wie IO in verschiedenen Anwendungen aussieht, beleuchtet dieser Abschnitt verschiedene Projekte und stellt deren typischen IO dar. Außerdem wurde auch mit den Entwicklern der Anwendung gesprochen, welche Ansatzpunkte diese für IO-Forwarding in ihrem Projekt sehen. Es wurde auch diskutiert, wie das IO-Forwarding am leichtesten eingepasst werden könnte und wo es in der Anwendung am effektivsten einsetzbar ist. Den Entwicklern wurden Lösungen zur Diskussion unterbreitet.

2.2.1 IBM Blue Gene/P

Am HLRS und weiteren Instituten werden Projekte durchgeführt, die man nach IO-Charakteristiken einordnen kann. Um IO optimieren zu können, ist es nützlich sich einmal die Art und Menge des IOs in verschiedenen Projekten anzuschauen. Daraus kann Schlüsse ziehen, auf welche Art der IO zu optimieren ist und wie der IO transportiert werden soll.
Ein Beispiel für die Herausforderung, die IO an ein Rechensystem stellt, ist Intrepid, das IBM Blue Gene/P System an der Argonne Leadership Computing Facility [19]. Es gehört zu den Top 5 der schnellsten Supercomputer von 2008. Die Eckdaten sind:

- 40960 Quad Core PowerPC 450 Rechenknoten mit jeweils 2 Gbyte RAM
- 640 Quad Core PowerPC 450 IO-Knoten mit jeweils 2 Gbyte RAM
- Anbindung mit 10 Gigabit Ethernet Switch
- 128 Dual Core Opteron Server als Speicherknoten mit jeweils 8 Gbyte RAM

- 16 DataDirect S2A9900 Kontrollerpaare mit 480 1 Tbyte-Platten-Speichersystem und 8 Infiniband Ports pro Paar.

Die Bandbreite zwischen Rechenknoten und IO-Knoten geht bis zu ca. 650 Mbyte/s. Die Bandbreite zwischen IO-Knoten und File Server geht bis zu ca. 250 Mbyte/s. Die Bandbreite pro Knoten zu den DataDirect-Speichersystemen geht bis zu ca. 600 Mbyte/s.
In der vorherig zitierten Referenz wird untersucht, wie die Lücke zwischen Rechenleistung bei diesem System und seiner IO-Leistung möglichst klein gehalten werden kann. Es wird gezeigt, wie spezifische Features des Dateisystems und von IO-Software helfen, eine hohe Performance anhaltend zu gewährleisten. Dabei wird auch die darunterliegende Hardware analysiert. Es wird festgestellt, daß das Intrepid IO-System sich gut an die Anforderungen von Large-Scale Applications anpassen läßt. Dies gilt jedoch nicht für eine sehr hohe Anzahl an Prozessen. Da das IO-System hier noch nicht das Maximum seiner Leistung zur Verfügung stellt, besteht noch Spielraum.
Aber es ist ungleich schwerer bis an das Maximum der Leistung des IO-Systems bei höchsten Anzahlen von Prozessen zu kommen. IO wird schnell der Flaschenhals für die Anwendung. Es wird gefolgert, das IO-System nahe am Maximum seiner Leistungsfähigkeit zu betreiben, um Läufe mit sehr großer Prozessanzahl besser bewältigen zu können. Die Ersteller von Anwendungen sollten sich der Problematik des IO-Systems bewusst sein und auch die Optimierungsmöglichkeiten und Tuning-Parameter von IO-Software nutzen.

2.2.2 Simulation mit Knochenmaterial

Ein weiteres Beispiel ist eine Simulation am HLRS, die im Zusammenhang mit biomechanischen Simulationsaktivitäten entwickelt wird [20]. Das Ziel ist es, das durchschnittliche elastische Verhalten von Spongiosa-Knochenmaterial innerhalb einer kontinuierlichen Skalierung, zu untersuchen. Dabei werden biomechanische Simulationen von Knochenimplantatsystemen durchgeführt, die durch klinische Computertomographie verfügbar sind. Die Simulationen sind direkte mechanische Simulationen der Spongiosa Knochenmikrostruktur. Die Geometrie von der Knochenmikrostruktur wird durch Micro-Focus-Computertomographien erhalten. Aus diesen detailierten Datensätzen und mit der Hilfe der implementierten Simulations Tool-Chain kann eine Transferfunktion zwischen klinischen CT-Bildern und dem Feld der inhomogenen, anisotropischen Verteilung der Knochen erhalten werden. Die Anwendung der Simulation Tool-Chain z.B. über einen Hüftgelenkskopf erfordert etwa 10000-20000 Ausführungen der involvierten Pre- und Post-Processing-Programme und 60000-70000 Läufe des benutzten Finite Element-Lösers.
Für die Simulation wird der Knochen in Würfel zerlegt, sogenannte „Representative Volume Elements“ (RVEs). Abbildung 2.1 auf der nächsten Seite zeigt den Anfall entsprechender Datenmengen für Simulation und der Vor- und Nachbereitung der Daten.
Beim Pre-Processing wird Input von einer Datei gelesen mit direkten Zugriff auf Blöcken von ca. 1 kByte. Der Output erfolgt in mehrere Dateien sequentiell in Blöcken von einigen Byte. Bei den folgenden Finite Element-Berechnungen erfolgt der Input sequentiell

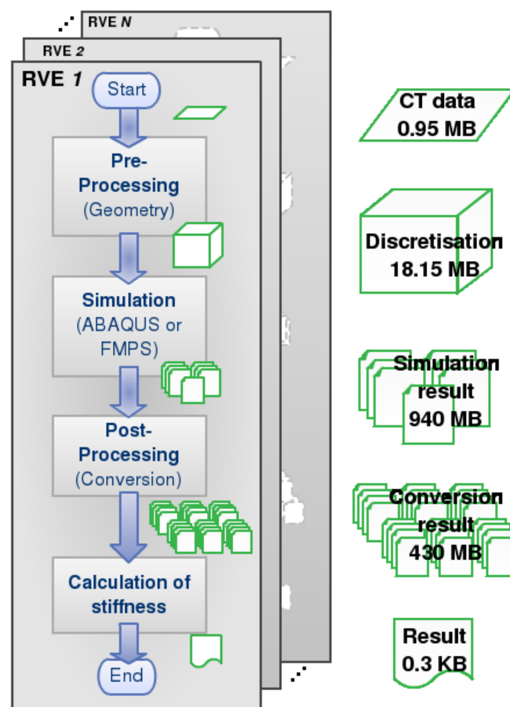

Abbildung 2.1: Biomechanische Simulation von Knochenimplantatsystemen und dabei anfallende Datenmengen

von der vorherigen Output-Datei zeilenweise. Der Output erfolgt wiederum in eine Datei in Blöcken von einigen Byte bis zu mehreren hundert kByte. Die darauf folgende Berechnung der Materialdaten nimmt die vorherige Output-Datei als Input in Blöcken von mehreren hundert kByte bis in die Größenordnung von Mbyte. Der darauf folgende Output in eine Datei hat ein Volumen von insgesamt einigen Hundert kByte. Abbildung 2.2 auf der nächsten Seite zeigt dies in der Übersicht.

2.2.3 Science Experimental Grid Laboratory

SEGL (Science Experimental Grid Laboratory) [21] ist ein Open Source Project am HLRS. SEGL ist ein Instrument zum Beschreiben, Validieren, Starten, Monitoring, Dispatching und Analysieren von großen wissenschaftlichen und industriellen High Performance Rechenexperimenten. Um die Experimente zu beschreiben wird GriCol (Grid Concurrent Language) verwendet. GriCol beschreibt das ganze Experiment auf oberster Ebene (Control Flow Level) als Satz von Blöcken und Links und mit Modulen mit Transitionen auf dem Data Flow Level. SEGL benutzt Wrapping Principles um einige Jobs auf High Performance-Computern auszuführen. Man hat ebenfalls die Möglichkeit Parameter-Studien durchzuführen, indem man den Job über ein Array von Data Sets ausführt. Man kann eine Liste von Eingabe-Dateien angeben und übereinstimmend mit

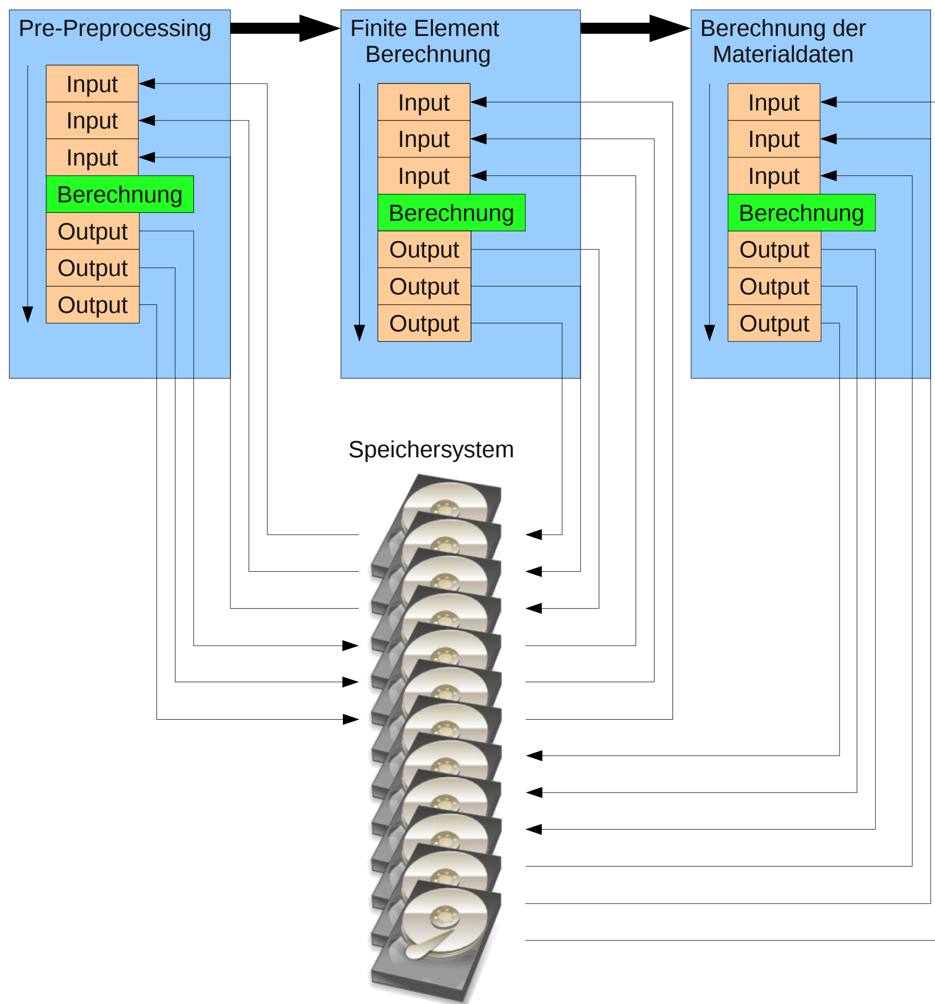

Abbildung 2.2: Biomechanische Simulation von Knochenimplantatsystemen mit Ablauf Input und Output von Dateien

den Eingabedateimasken wird SEGL bestimmen wieviele Läufe durchgeführt werden. Man kann auch eine feste Anzahl von Data Sets angeben.
SEGL [22] besteht aus 3 Hauptkomponenten:

- User Workstation (Client)
- ExpApplicationServer (Server)
- ExpDBServer (OODB).

Das System operiert als Client-Server-Modell innerhalb dessen der ExpApplicationServer mit dem entfernten Ziel-Computer unter Benutzung eines Grid Middleware Services

interagiert. Der ExpDBServer (OODB) ist realisiert mit der Implementierung von Java Data Objects (JDO). Jedes Experiment innerhalb von SEGL ist auf 3 Ebenen beschrieben:

- Control Flow von Compute Block (Beschreibung eines logischen Schemas des Experiments)
  - Richtung
  - Bedingung
  - Sequenz
- Data Flow (lokale Beschreibung von interblock Computation-Prozessen)
- Data Repository.

Wesentlich für den IO von Daten innerhalb von SEGL sind

- Input Set von Daten mit den Parametrierungsregeln
- Input und Output von Daten von/an die Compute Resourcen.

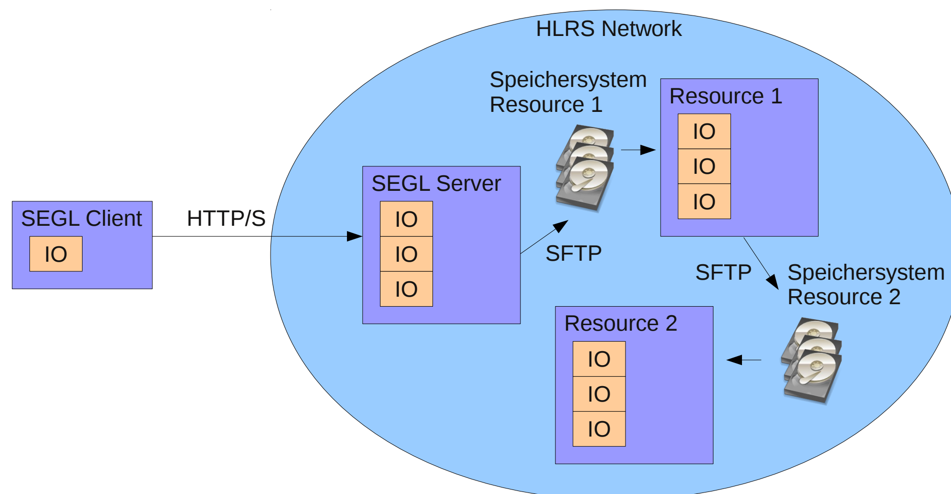

Abbildung 2.3: Transport von Daten innerhalb von SEGL

Die Weiterleitung von Daten zwischen den Compute Resourcen (siehe Abbildung 2.3) könnte auch mit einem speziellen Dateisystem erfolgen, das die Daten entgegennimmt und den Transport optimiert. Hier könnten viele Dateien entgegengenommen werden und mit einem einzigen Schreibvorgang übertragen und auf der Gegenseite wieder in mehrere Dateien aufgeteilt werden. Hierzu wäre eine Steuerdatei notwendig, die zumindest angibt, welche Dateien zu einer vollständigen Übertragung gehören.

2.2.4 Automatische Formoptimierung von hydraulischen Maschinen

Ein weiteres Beispiel einer Simulation ist die „Automatische Formoptimierung von hydraulischen Maschinen mit OpenFOAM“ am Institut für Strömungsmechanik und hydraulische Strömungsmaschinen in Stuttgart. Der Optimierungsprozess bei der Formoptimierung sorgt dafür, daß sich die Berechnungszeiten verkürzen. Hier werden viele Läufe parallel durchgeführt (momentan bis zu 72 Läufe) und dabei wird die beste Optimierung nach einem vorgegebenen Kriterium ausgewählt. Hierbei werden robuste Berechnungsschemata verwendet, die zu einem verwertbaren Ergebnis führen. Die Optimierung hat mit geringem Aufwand hohen Nutzen und benötigt auch nur geringfügig Personal zur Anwendung. Es können verschiedene Design-Kriterien gesetzt werden (Partload, BEP, Overload) und es ist möglich mehrere Parameter für die parallelen Durchläufe gleichzeitig zu setzen. Abbildung 2.4 zeigt den Workflow der Simulation und die dazugehörigen Datenmengen. Die Daten werden alle sequentiell und zeilenweise aus der Datei transfe-

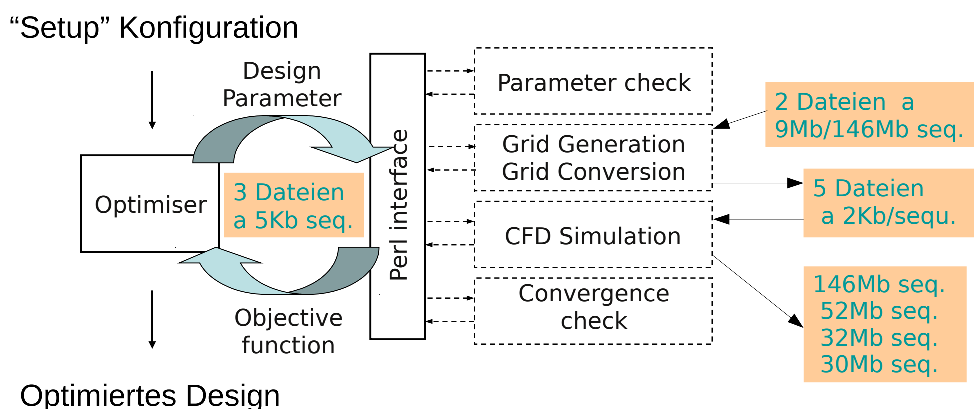

Abbildung 2.4: Workflow automatische Formoptimierung von hydraulischen Maschinen mit OpenFOAM

riert. Daher sind die Datenmengen pro Transferanweisung (read/write) eher klein (im Byte-Bereich). Während des Workflows werden kleinere Datenmengen zunächst zur Vorbereitung der eigentlichen CFD Simulation geschrieben/gelesen. Eine Optimierung des IO ist beim Schreiben der Simulationsdaten empfehlenswert. Hier werden größere Datenmengen in Dateien abgelegt. Der Optimierer benötigt auch nur kleine Datenmengen, die von Dateien gelesen/geschrieben werden. Auch hier bringt eine Optimierung des IO keine wesentlichen Vorteile. Die angegebenen Datenmengen gelten exemplarisch für einen einzelnen Lauf von mehreren parallel stattfindenden Läufen.

2.2.5 Distributed European Infrastructrure for Supercomputing Applications

Das DEISA (Distributed European Infrastructrure for Supercomputing Applications) Projekt [23] verfolgte die Entwicklung einer verteilten High Performance Computing-Infrastruktur in Europa. Ein Hauptziel ist die übergreifende Zusammenarbeit von nationalen Rechenzentren in Europa. Die DEISA Infrastruktur ist weltweit unerreicht in ihrer Heterogenität und Komplexität. Für die Übertragung von größeren Datenmengen zwischen den Rechenzentren ist das DEISA-Netzwerk (Abbildung 2.5) zuständig. Die Daten werden hier also auch über eine WAN (Wide Area Network)-Verbindung

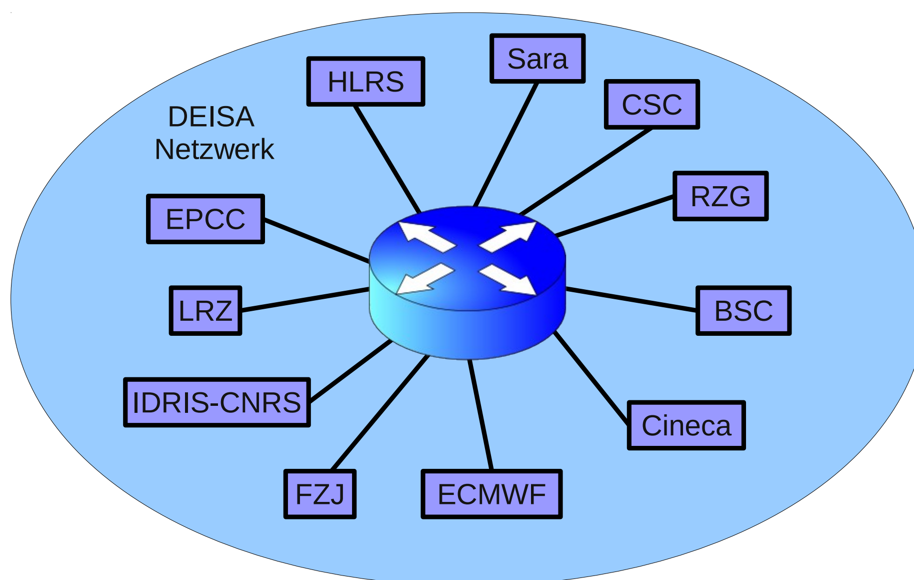

Abbildung 2.5: DEISA-Netzwerkinfrastruktur mit 10Gbit-Anbindung

vom/zum Speichersystem des entfernten Rechenzentrums transferiert. Innerhalb des DEISA-Netzwerkes wird hauptsächlich das Dateisystem GPFS als paralleles Dateisystem verwendet und auch standortübergreifend eingesetzt. Der Austausch von Daten zwischen den Rechenzentren erfolgt über das parallele Dateisystem oder über GridFTP. Die GridFTP Memory-to-Memory Transferrate geht bis zu etwa 1 Gbyte/s. Die IO-Forwarding Software IOFWD (Kapitel 3 auf Seite 21) wurde schon testweise installiert. Jedoch gab es noch Probleme mit der Aktualität der Software zum automatischen Erstellen der Build-Umgebung für das Kompilieren von IOFWD. Ein ordnungsgemäßes Kompilieren von IOFWD war so nicht möglich.

2.2.6 Große Multiskalen-Simulation

In Kapitel 7 auf Seite 85 wird IO-Forwarding in eine große Multiskalen-Simulation im aero-akustischen Bereich [24], die erhebliche Datenmengen erzeugt, eingebunden. Die aero-akustische Simulation findet in einem heterogenen Super Computing-Umfeld statt. Die integrierte Simulation von Flüssigkeitsströmung mit ihrer Aero-Akustik ist eine typische Multiskalen-Simulation mit unterschiedlichen numerischen Anforderungen in den involvierten Teilen. Die unterschiedlichen Anforderungen können räumlich getrennt werden, da das schall-generierende Objekt generell ziemlich klein ist, wenn man es mit dem Gebiet vergleicht, das für die Schallausbreitung berechnet wird. Eine natürliche Aufteilung in ein Gebiet der Schallerzeugung und ein Gebiet der Schallausbreitung liegt nahe. Beide Gebiete sind nur schwach mit numerischen Anforderungen gekoppelt, die sich stark voneinander unterscheiden. Die heterogenen Rechengebiete sind den geeigneten Architekturen zugeordnet. Der betreffende IO wird auch in Kapitel 7 auf Seite 85 näher dargestellt.

3 Realisierung IO-Forwarding mit IOFWD

3.1 IO-Forwarding

Um IOFWD zu verstehen, wird zunächst erklärt, wie IO-Forwarding funktioniert. Abbildung 3.1 zeigt ein Beispiel anhand eines write Befehls. Hier wird vereinfacht dargestellt,

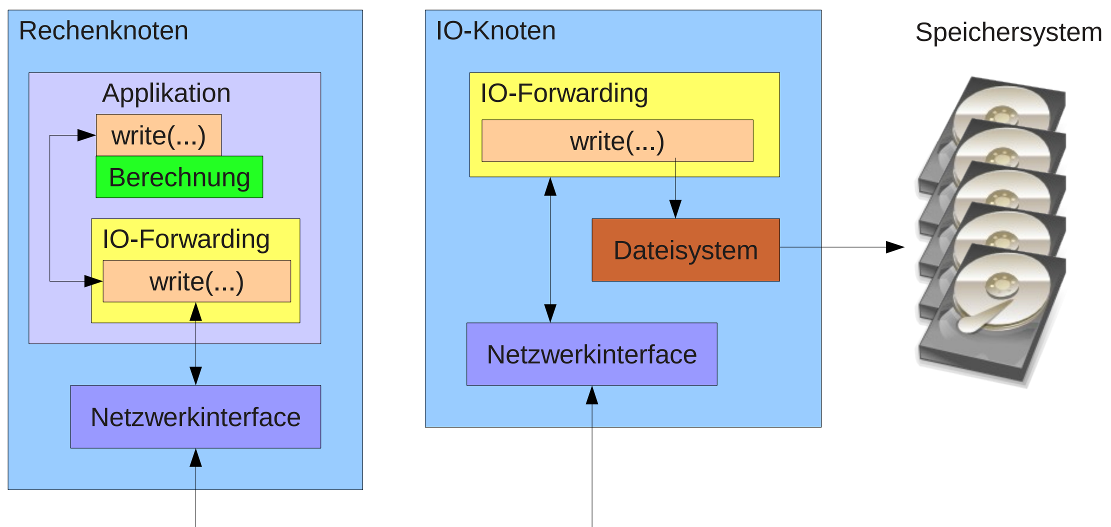

Abbildung 3.1: Übersicht IO-Forwarding: Weiterleitung eines Schreibbefehls

wie der IO-Aufruf im Rechenprozess abgefangen und an den IO-Server weitergeleitet wird. Der Rechenprozess ist gleichzeitig der IO-Client. Der write IO-Befehl wird direkt im Speicher abgefangen und über das Netzwerk zum IO-Knoten verschickt, der gewöhnlich direkt der IO-Server ist. In den nachfolgenden Abschnitten werden Dateisystem und Plattensystem weggelassen, sie schliessen sich jedoch immer den Operationen auf dem IO-Server an.

3.2 GASNet

IO-Forwarding braucht eine Transportschicht, die einfach und logisch zu bedienen ist. Der IO der Applikation wird dann hierüber transportiert.

3.2.1 Aufbau und Relation zu IOFWD

GASNet unterstützt IOFWD bei der Aufgabe des IO-Forwarding. GASNet besteht im wesentlichen aus drei Teilen:

- Extended API
- Core API
- Netzwerk-Conduit.

Wobei alles zunächst auf dem Netzwerk-Conduit aufbaut. Hier wird anhand eines Protokolls wie UDP oder Infiniband bestimmt, wie die Daten aus dem Core API transportiert werden. Das Core API stellt bestimmte Nachrichtenformen wie Request-Reply Sequenzen zur Verfügung mit denen Inhalte über den Netzwerk-Conduit transportiert werden können. Das Core API erfordert mehr Koordination der Nachrichten als das Extended API. Das Extended API mit seinen put/get-Sequenzen baut auf dem Core API auf und ist komfortabler. Auf dieser logischen Ebene sind auch Funktionen des IOFWD API angeordnet. Diese wurden aus dem Core API zusammengestellt. Ein Teil der Funktionen von IOFWD speziell die put/get Sequenzen sind somit eine Erweiterung des Extended API von GASNet.

3.2.2 Funktion

Von der Applikation auf dem Rechner werden während ihrer Laufzeit Daten wie Variableninhalte in den Speicher geschrieben und vom Speicher gelesen. Der Speicher ist für gewöhnlich der lokale Speicher des Rechners. Möchte nun die Applikation z.B. Variableninhalte in den Speicher eines anderen Rechners schreiben, ist es erforderlich diese Daten über ein Netzwerk, das die Rechner miteinander verbindet, zu transportieren. Der herkömmliche Weg (z.B. in MPI) ist es, explizit einen Nachrichtenaustausch über das Netz zu programmieren, der dies bewerkstelligt, in dem auch die Daten übertragen werden. Hierbei ist auch gewöhnlich Kenntnis über die Datenstruktur erforderlich. Mittlerweile existieren Software Frameworks wie GASNet, die es ermöglichen, Daten mit einem einfachen put/get Befehl in eine bestimmte Speicherstelle vom entfernten Rechen-Knoten zu senden/empfangen. Hierbei sind gewisse Speicherstellen von einem Rechner in einem Netzwerk allen anderen Rechnern bekannt. Man spricht hier von einem globalen Speicher-Adressraum („Global Address Space“), da für jeden Rechner im Netzwerk Speicherstellen aller anderen Rechner sichtbar sind. Sichtbar bedeutet, es kann auf diese entfernten Speicherstellen gelesen/geschrieben werden.

Die Interpretation der Inhalte der Speicherstellen, die übertragen werden, wird von dem Software-Framework wie GASNet nicht vorgenommen und der darüberliegenden Schicht überlassen. Spielt die Tatsache, ob eine Speicherstelle lokal auf dem Rechner liegt, eine Rolle, spricht man von „Partitioned Global Address Space - PGAS“.
In dieser Arbeit wird das Global Address Space Framework GASNet verwendet, um dateibezogene Daten zwischen Speicherstellen unterschiedlicher Rechner zu verschieben. Dazu wird GASNet modifiziert und angepasst.

3.2.3 Request und Reply

Die Request-Reply Sequenzen sind Active Messages [25]. Abbildung 3.2 zeigt, wie die Request-Reply Sequenz in IOFWD benutzt werden. Die maximale Länge eines Nach-

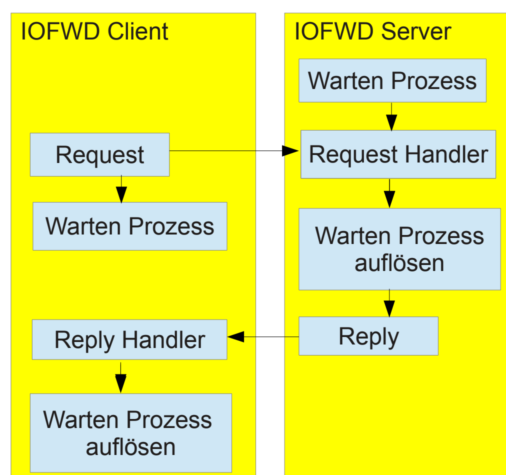

Abbildung 3.2: Von IOFWD benutzte Request-Reply Sequenz

richtenaustausches ist eine Request-Reply Sequenz mit den involvierten Handlern. Aus den Request-Reply Sequenzen des Core APIs wurden put/get Sequenzen im Sinne des Extended APIs in IOFWD gebaut. Request bzw. Reply Nachrichten gibt es in drei Formen:

- Short (ohne Payload)
- Medium (mit mittlerem Payload)
- Long (mit großem Payload).

In IOFWD werden Short und Long Nachrichten für die put/get Sequenzen verwendet. Es existiert eine put Funktion für Kommandos und eine put und get Operation für den Transport von Dateiinhalten. Abbildung 3.3 auf der nächsten Seite zeigt die Auflösung

von jeweils put und get für den Transport von Dateiinhalten für die read bzw. write IO-Funktionen. Die Requests vom Typ Short werden so aufgerufen, daß sie die Adressen des

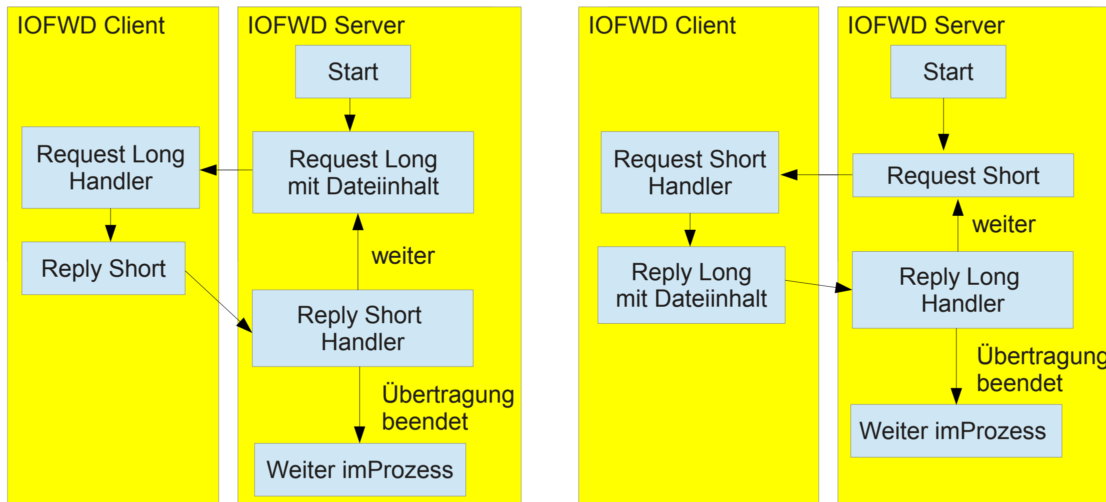

Abbildung 3.3: Links Operation put und rechts Operation get für Dateiinhalte

zu lesenden/schreibenden Puffers mit Längenangabe schicken. Put und get sind jeweils aus Sicht des Servers zu sehen. Die put Funktion für IO-Aufrufe (Kommandos wie open, read, write, close) entspricht im wesentlichen Abbildung 3.2 auf der vorherigen Seite mit Requests des Typs Long. Dieses wird später noch genauer ausgeführt. Generell wird erst ein put für IO-Kommandos abgeschickt, welches im Falle von read und write Funktionen ein put/get für Dateiinhalte nach sich zieht.

3.2.4 Heterogenität

GASNet ist ausgelegt für homogene Netzwerkstrukturen, im speziellen für homogene Rechennetze. Ein bisher wenig betrachtetes Gebiet ist die Anwendung und Anpassung dieser GAS Frameworks auf heterogene Netzwerke. Als wichtige Anwendung wird hier IO-Forwarding auf Basis eines heterogenen GASNet Netzwerkes betrachtet. Dabei gibt es zwei unterschiedliche Knotensorten im GASNet Netzwerk:

- Rechenknoten
- IO-Knoten.

Rechenknoten und IO-Knoten unterscheiden sich im betrachteten Fall auch hinsichtlich des Betriebssystems und der Hardware. Die folgenden Abschnitte beschreiben auch diese Anpassungen. Diese Anpassungen betreffen auch IOFWD.

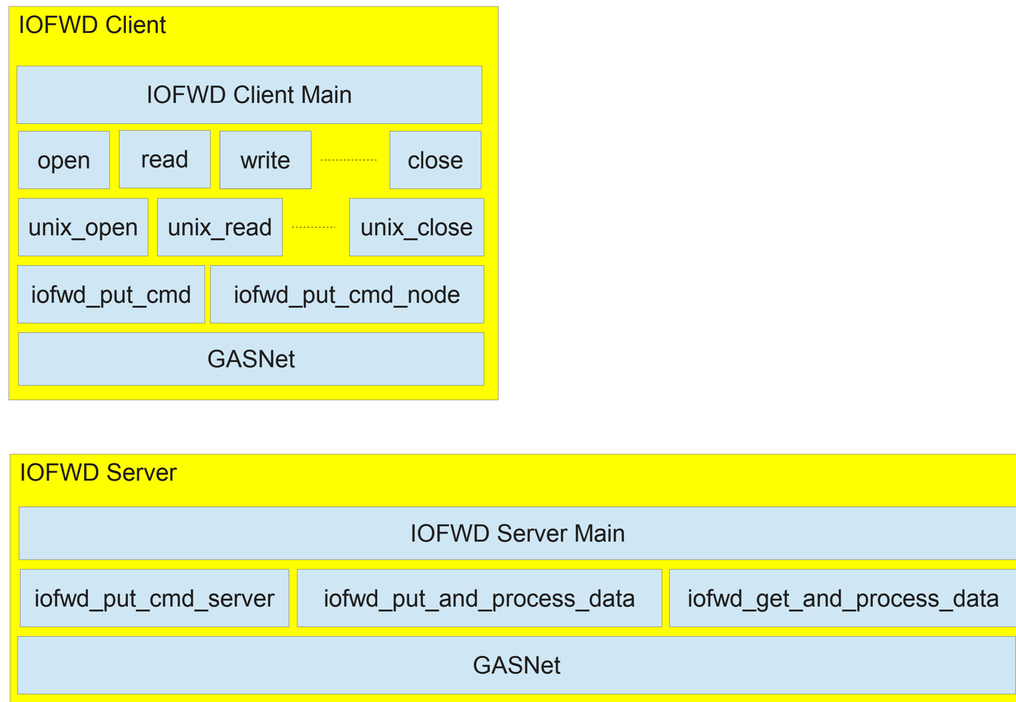

Abbildung 3.4: get/put Layer

Die Anpassung, die alleinig GASNet betreffen, sind das Anlegen und die Dimensionierung der Transfersegmente zum Überspielen der IO-Daten von einem Prozess zum anderen. Es wurde eine Aushandlung einer gemeinsamen Speicherseitengröße über alle Systeme hinzugefügt. Es wurde davon ausgegangen, daß kleinere Speicherseiten ganz in größere Speicherseiten aufgehen. Weiterhin ist das Anlegen von Funktionspuffern in der Dimensionierung berücksichtigt worden. Bei heterogenen Systemen muß darauf geachtet werden, daß die Transferpuffer mit allen Unterpuffern kongruent verteilt sind. Diese Anpassungen sind gemacht worden. Im Moment gibt es in GASNet einen Hauptpuffer mit zwei Unterpuffern. Der eine ist der GASNet Transferpuffer und der andere ist ein Puffer für Funktionen und besitzt nur die Möglichkeit eines indirekten Zugriffs. Dieser wird nicht von IOFWD benutzt. Es wird nur der Transferpuffer benutzt. Dieser wird unverändert in IOFWD übernommen d.h. GASNet kümmert sich komplett um die Dimensionierung dieses Puffers. Diese Dimensionierung ist komplett in GASNet angepasst worden. Der Transferpuffer wird in IOFWD für den Threaded Client unterteilt in Unterpuffer pro zu bedienenden IOFWD Server.

3.3 Kategorisierung der Aufrufe

Es werden 44 IO-Funktionen von IOFWD unterstützt. Die IO-Funktionen lassen sich in verschiedene Bereiche einteilen.

3.3.1 Öffnen und Schliessen

Zunächst gibt es die Funktionen

- open
- close.

Diese Funktionen öffnen bzw. schliessen eine Datei. Damit eine Datei beschrieben/gelesen werden kann, muß diese vorher mit open geöffnet werden. Dies wird gemacht, um eine einmalige Vorprüfung zu machen, ob eine Datei aus dem Dateisystem für weitere Operationen verfügbar gemacht werden kann. Viele IO-Operationen benutzen statt eines Dateinamens den File Deskriptor der Datei, den open zur Verfügung stellt. Ist dies der Fall, braucht dies bei späteren IO-Operationen nicht mehr geprüft zu werden. Nach Abschluss aller Schreib- und Leseoperationen, üblicherweise wenn die Datei nicht mehr gebraucht wird, wird das close Kommando verwendet. Dieses Kommando wird natürlich auch auf den File Deskriptor angewendet.

3.3.2 Schreiben und Lesen

Dann kommt die Gruppe von IO-Aufrufen, die im Umfeld von Schreib- und Leseoperationen liegen:

- read
- write
- pread
- pwrite
- pread64
- pwrite64
- lseek
- readv
- writev.

Alle Funktionen außer lseek benötigen einen Puffer zum Lesen/Schreiben von Dateiinhalten. In diesen Puffer schreibt/liest die Funktion rein/raus. Die lseek Funktion positioniert den Schreib-/Lesepunkt in einer Datei. Der Lesepunkt kann relativ zum Anfang/Ende oder zur aktuellen Position verschoben werden. Die Operation wird später wichtig bei der parallelen Verwendung von Operationen in Threads. Die pread/pwrite Funktionen verwenden einen eingebauten Positionierer d.h. lseek wird hierüber mit eingeschränkten Eigenschaften angesteuert. Zu den pread/pwrite Funktionen gibt es auch 64 bit-Varianten. Die Seek-Operationen sind dann mit 64 bit weiten Bereichen möglich. Die Funktionen readv/writev schreiben/lesen multiple Puffer, welche als Vektoren dargestellt sind. Diese Funktionen sind die komplexesten Funktionen bezüglich des Schreibens/Lesens einer Datei. Üblicherweise werden read/write Operationen verwendet. Daher tauchen diese später in den Beispielen auf. Sie reichen völlig aus, um die Funktion von IOFWD zu erläutern.

3.3.3 Meta-Daten und Ordner

Die Funktionen, die sich mit der Manipulation von Metadaten der Dateien bzw. Ordner beschäftigen, werden im Folgenden genannt:

- access
- chmod
- fchmod
- stat
- fstat
- lstat
- stat64
- fstat64
- lstat64
- utime
- chdir
- fchdir
- mkdir
- rmdir
- getcwd

- chown
- fchown
- lchown
- getdents
- getdents64.

Hierbei handelt es sich für die Funktionen access, chmod und fchmod um die Darstellung und Veränderung von Rechten einer Datei bzw. eines Ordners. Ein Ordner kann man als Datei sehen, in die man nicht lesen/schreiben kann. Mit den Operationen stat, fstat, lstat, stat64, lstat64 und utime lassen sich Datei oder Dateisysteminformationen anzeigen. Mit chdir, fchdir, mkdir, rmdir und getcwd lassen sich Ordner manipulieren. mkdir hat noch zusätzlich die Möglichkeit chmod zu steuern. chown, fchown und lchown können die Besitzrechte ändern. Man unterscheidet zwischen Zugriff über den Pfad und über den File Deskriptor. Die Operation getdents liest spezielle Ordnerstrukturen aus.

3.3.4 Datei-Verknüpfung

Dateien oder Ordner lassen sich mit weiteren Dateien oder Ordnern verknüpfen. Nachstehende Operationen kümmern sich um Verknüpfungen im Dateisystem:

- link
- symlink
- unlink
- readlink
- rename.

File Deskriptoren werden nicht verwendet, sondern Pfadnamen. Mit diesen Operationen ist es möglich, mit mehreren Dateien auf gleiche Inhalte zu verweisen. Hierbei wird natürlich auch die Funktion getcwd angewandt, damit das Working Directory auch auf dem Server gesetzt wird im Falle von nicht absoluten Pfaden, da der Befehl ja auch mit nicht absoluten Pfaden aufgerufen wird.

3.3.5 Verschiedenes

Diese Operationen führen unterschiedliche Vorgänge aus, die sich nicht direkt eingruppieren lassen:

- truncate
- truncate64

- ftruncate
- ftruncate64
- fsync
- fcntl
- statfs
- statfs64.

Es fällt hier auch auf, daß es hier auch wieder 64 bit-Varianten gibt. Die truncate Funktionen kürzen Dateien, wobei die ftruncate Funktionen wieder auf File Deskriptoren arbeiten. Die fsync Operation synchronisiert den Status der Datei mit einem Speichersystem wie der Festplatte. Diese Operation wurde bei FUSE-Erweiterung von IOFWD angewendet, um sicherzustellen, dass Inhalte auch wirklich geschrieben worden sind. Die fcntl Funktion erwies sich als schwierig, da sie einige Fehler verursachte. Die Fehler wurden in zahlreichen Versuchen und Tests erkannt und behoben.

3.4 Grundlegende Informationsstruktur

In Abbildung 3.5 sieht man die grundlegende Informationsstruktur von IOFWD.
Grundlegend gibt es einen Teil, der innerhalb IOFWD eine Bedeutung hat und einen Teil

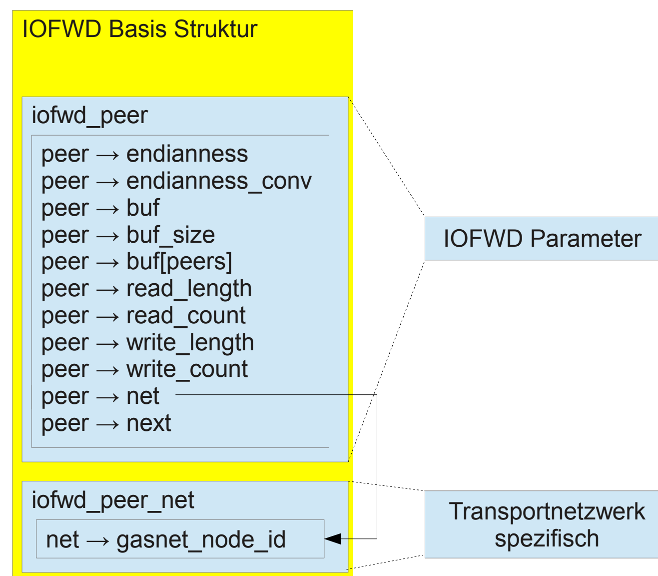

Abbildung 3.5: IOFWD Basis-Struktur

der spezifisch für das darunterliegende Transportnetzwerk ist. Das verwendete Transportnetzwerk ist hier GASNet. Der Grund für eine Separierung der Struktur ist, das es sich gezeigt hat, dass eine Abstrahierung in diesem Fall schlecht möglich ist bzw. keinen Sinn ergibt. Die Parameter des Transportnetzwerkes können sehr unterschiedlich sein. BMI und GASNet haben dies gezeigt. Die Besonderheit hier ist auch, dass eine Besetzung der Parameter im transportnetzwerk-spezifischen Teil, erst mit der Initialisierung des Transportnetzwerks erfolgt. Bei der Anlage solcher Strukturen, die erst später genauer definiert werden, spricht man von unvollständigen Typen. Auch aufgrund dieser Vorgehensweise ist eine Abstrahierung dieses Teils der Basisstruktur schwer möglich. Die komplette Informationsstruktur existiert für jeden Knoten. Die Informationsstrukturen sind in einer einfach verketteten Liste abgelegt. Es befinden sich Informationen darin zu:

- Endianness
- eigene IOFWD-Transferpufferadresse und Größe
- IOFWD-Transferpufferadressen aller anderen Knoten
- Anzahl von zu puffernden Lese- und Schreibblöcken
- Transportnetzwerkspezifika wie Knotenadresse.

3.5 Endianness

Die Endianness eines Systems legt fest, wie Datentypen, die sich über mehrere Byte erstrecken im Speicher des Rechenknotens abgelegt werden. Die Reihenfolge der Bytes

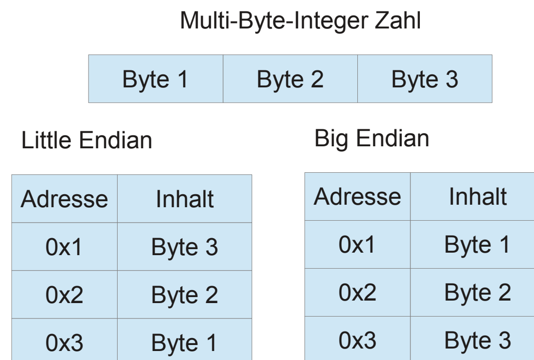

Abbildung 3.6: Endianness

ist entscheidend. Abbildung 3.6 zeigt die Darstellung für die beiden verwendeten Datentypen „Little Endian“ und „Big Endian“. Die SX-Knoten sind Big Endian und die

Linux-Knoten sind Little Endian-Systeme.
Es läßt sich so am System erkennen, welche Endianness vorliegt. IOFWD erkennt, welche Endianness das eigene System hat und schickt diese Information an alle anderen Knoten. Daraus berechnet sich jeder Knoten, ob für das Zielsystem konvertiert werden soll oder nicht. Die Information wird in der Basisstruktur abgelegt (Abbildung 3.7). Abbildung 3.8 auf der nächsten Seite zeigt den Vorgang der Endianness-Konvertierung

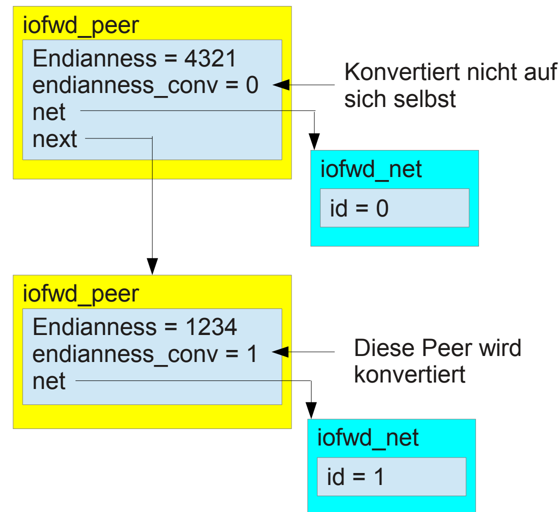

Abbildung 3.7: Endianness-Struktur

in IOFWD. Es wird in IOFWD nur auf Little Endian-Systemen konvertiert, da die Konvertierung unvorteilhaft für die verwendeten Big Endian-Systeme mit Vektor-CPU ist. Die Konvertierung erfolgt durch kreuzweises Vertauschen der Bytes.

3.6 Initialisierung

Die Initialisierung von IOFWD erfolgt durch eine Abfolge der Transportnetzwerk-spezifischen Startprozesse und der Startprozesse von IOFWD selbst. Abbildung 3.9 auf Seite 33 zeigt die Hauptschritte der Initialisierung. Die GASNet Handler Definition legt die Call-Back Funktionen der Gegenseite fest, d.h. hier wird bei einem Request von einem Knoten an einen anderen Knoten der entsprechende zu diesem Zweck registrierte Handler aufgerufen. Der Handler ist eine für IOFWD erstellte Funktion, die auch den Inhalt von Elementen der IOFWD Basisstruktur setzt. Mit dem Handler werden auch Wartevorgänge wieder aufgelöst, d.h. es wird ein Signal zum Weitermachen gesetzt. Der Client Name und der Server Name von IOFWD werden von außen über die Umgebung gesetzt. Diese Technik wird auch bei den Batch-Skripten für MPI Start-Umgebungen eingesetzt. Da der IOFWD Server vom IOFWD Client aus über SSH gestartet wird, ist auch ein

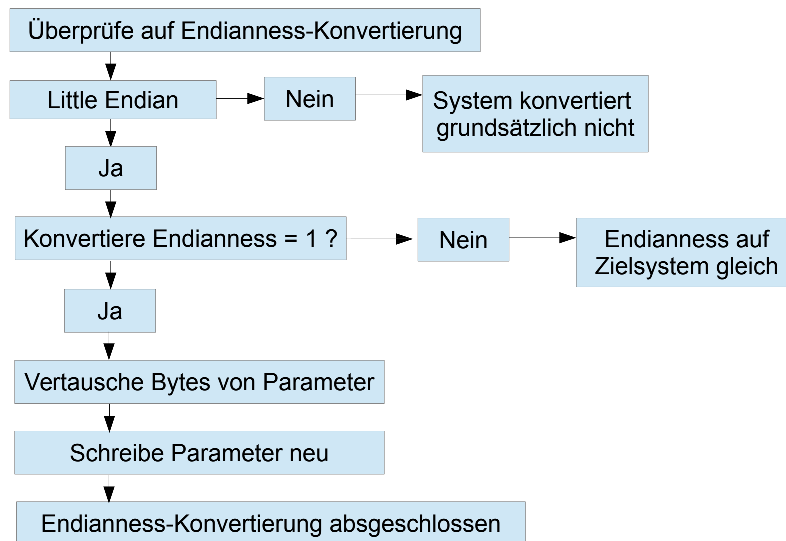

Abbildung 3.8: Endianness-Konvertierung

Pfad für das entfernte Binary des IOFWD Server mit anzugeben. Der GASNet Init-Prozess sorgt dafür, dass die Transfersegmente zur Verfügung gestellt werden und alle Segmente jedem Knoten bekannt sind. Es wird auch hier im Rahmen der Heterogenität die größte Speicherseitengröße aller Knoten zum Berechnen der Transfersegmente genommen. Diese Berechnung wurde nachträglich in GASNet vom Autor weiterentwickelt. Das Füllen der IOFWD Basis-Struktur wird später noch ausführlich behandelt. Ist die Initialisierung abgeschlossen, kann das IO-Forwarding eingeschaltet werden. Wie IOFWD die Einschaltung des IO-Forwarding genau behandelt, wird noch erklärt. Es wird hier die Identität, ob es sich um Server oder Client handelt, festgestellt. Der IOFWD Server ist natürlich ein komplettes Programm mit Hauptschleife (Abbildung 3.10 auf Seite 34) und unterscheidet sich in dieser Hinsicht vom IOFWD Client, der Aufrufe aus der Benutzeranwendung entgegennimmt. In diesem Sinne ist die Hauptschleife des IOFWD Clients natürlich die der Anwendung. Die Anfrage in der Hauptschleife des IOFWD Servers wegen des Signals wird von einem Handler beeinflusst, der Anfragen vom IOFWD Client entgegennimmt. Beim Bereitstehen des IO-Kommandos und weiterer Inhalte auf dem IOFWD Server wird das Signal freigeschaltet und die Hauptschleife schreitet fort. Es wird aus einem Array von IO-Funktionen die passende IO-Funktion auf dem IOFWD Server ausgewählt und mit den angelieferten Daten vom IOFWD Client aufgerufen. Das zurückzusendende Ergebnis mündet übrigens auch in ein put_cmd Kommando, das wiederum denselben Kommando-Handler auf der IOFWD Client-Seite aufruft, wie er bereits auf der IOFWD-Seite zum Weiterführen und Datenversorgung der Hauptschleife verwendet wurde. Der IOFWD Client braucht jedoch kein IO-Kommando zur Verfügung zu stellen. Hier wird nur der Return-Wert einer IO-Funktion entgegengenommen. Daher

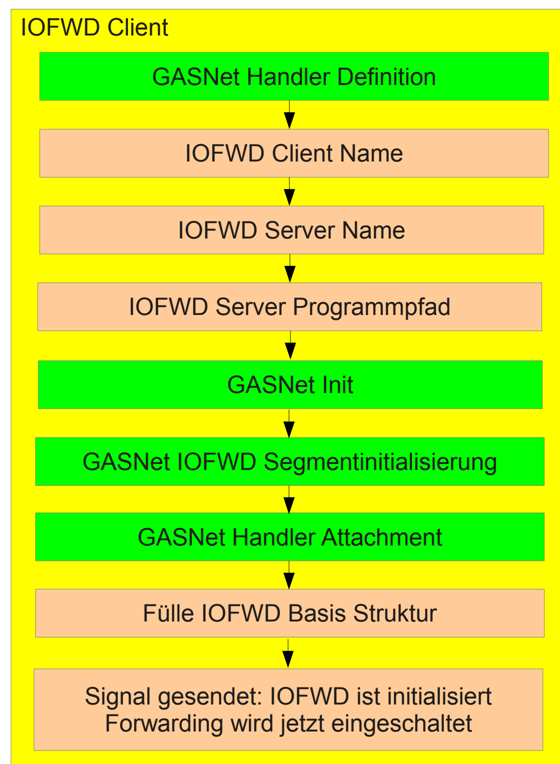

Abbildung 3.9: IOFWD Initialisierung: Übersicht

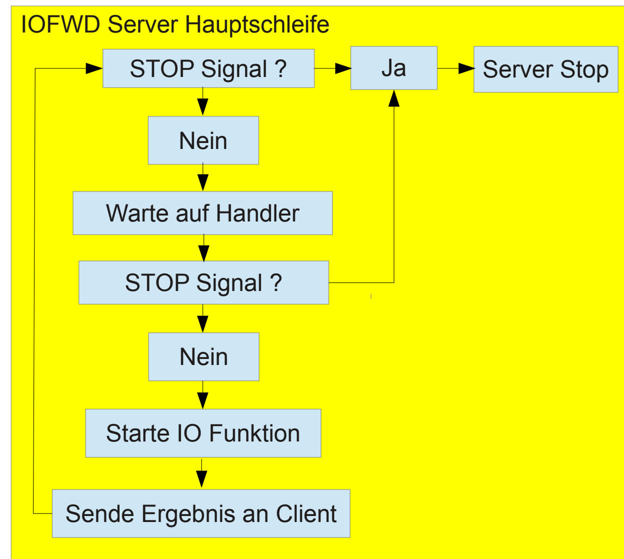

Abbildung 3.10: IOFWD Server Hauptschleife

wird mit dem Senden des Ergebnisses ein spezielles Kommando gesendet, das keinen Aufruf von IO-Funktionen zur Folge hat.
Das Füllen der IOFWD-Basisstruktur ist ein Teil der Initialisierung. Diese wird in Abbildung 3.11 auf der nächsten Seite nochmals detailliert dargestellt. Der Austausch der GASNet spezifischen Parameter ist hier nicht detailliert aufgeführt. Der User Block Transfer definiert wie viele Blöcke einer Datei zum Schreiben/Lesen gespeichert werden sollen, bevor sie gemeinsam geschrieben/gelesen werden. Dies setzt voraus, dass die Blöcke konsekutiv geschrieben/gelesen werden können. Der User-Block Transfer kann vom Benutzer verändert werden. Es handelt sich hierbei auch um die Tatsache, daß das Schreiben/Lesen vieler kleiner Blöcke administrativ sehr aufwendig ist. Das Schreiben/Lesen eines großen Blockes bedeutet deutlich weniger Kommunikation, Warten und IO-Verwaltungsroutinen. Gezeigt wird auch der Austausch der Endianness-Informationen und die Koordination des Austausches. Die Endianness-Informationen werden in die Basis-Struktur eingetragen. Die Kommunikation zwischen den Knoten erfolgt über Requests und zugeordnete Handler. Dabei werden Endianness-Information durch Handler gesetzt und auch Wartepunkte aufgelöst. Die Wartepunkte betreffen die Bereitstellung und die Sendebereitschaft von Endianness-Informationen.
Ein wichtiger Teil der Initialisierung ist der richtige Zeitpunkt zum Einschalten der IO-Forwardings von IOFWD. Während der Initialisierung von IOFWD werden auch IO-Kommandos wie open von IOFWD und GASNet selbst aufgerufen. Hier besteht die Gefahr, das IOFWD mit sich selbst interagiert und ein Endlos-Loop entsteht. Deshalb wird wie in Abbildung 3.12 auf Seite 36 vorgegangen. IOFWD leitet alle IO-Aufrufe wäh-

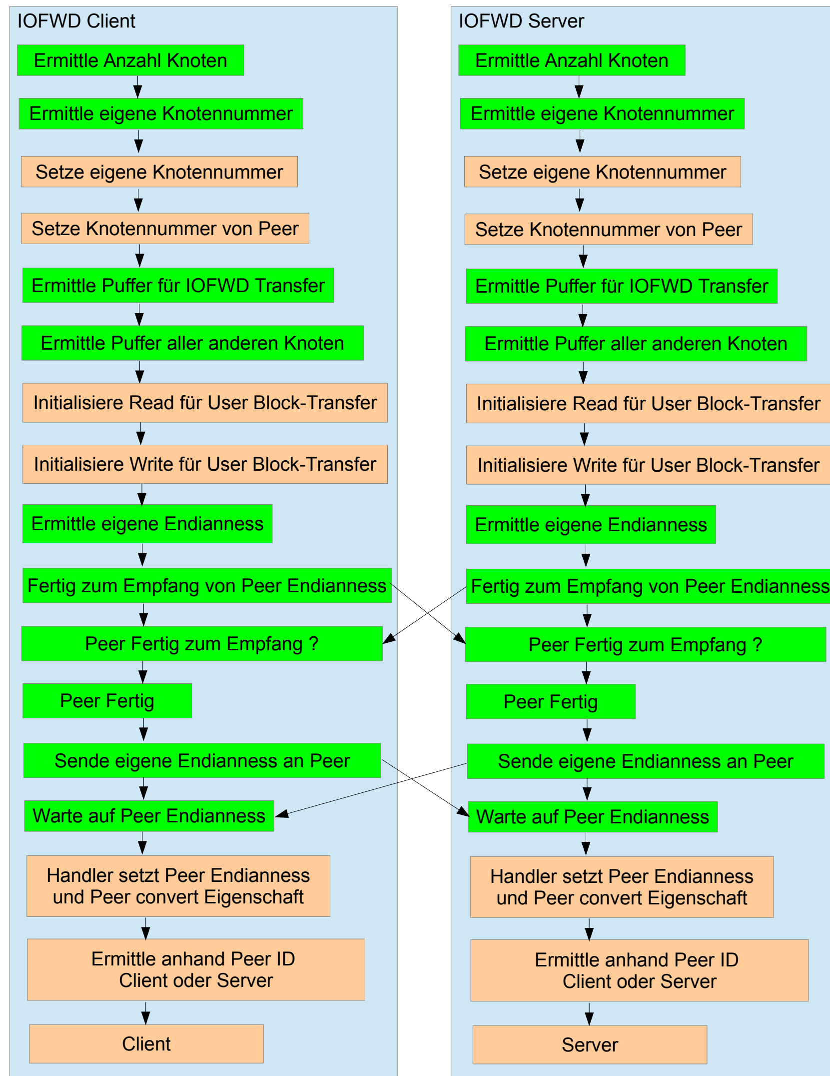

Abbildung 3.11: IOFWD-Initialisierung: Füllen der Strukturen

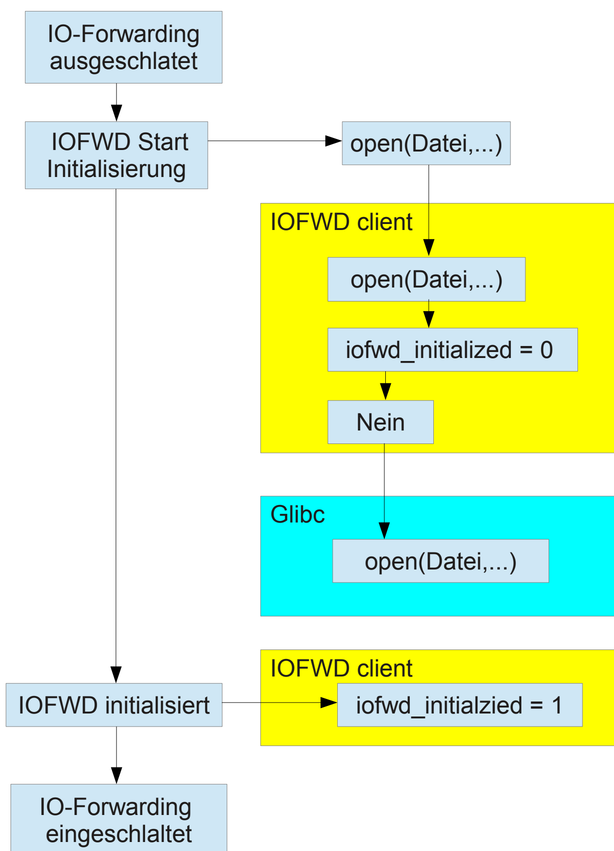

Abbildung 3.12: IOFWD Initialisierung: Einschalten des IO-Forwarding

rend der Initialisierung an die Original-Bibliotheksfunktion in der C-Systembibliothek weiter. Sobald alle Initialisierung abgeschlossen ist, wird das IO-Forwarding eingeschaltet. Es kann festgestellt werden, daß IOFWD und GASNet nicht mit sich selbst loopen, wenn das IO-Forwarding eingeschaltet wird. Dies liegt auch an dem Pfad-Mechanismus von IOFWD. Es wird anhand des Pfades sichergestellt, daß wirklich nur Aufrufe, die für IOFWD bestimmt sind, weitergeleitet werden. Diese Einschränkung ist nötig, da erstens ein Unterscheidungsmerkmal für IO-Forwarding einer Art getroffen werden muß und zweitens ein unübersichtliches Loopen mit sich selbst verhindert werden muß. Die Pfad-Selektion ist einfach umzusetzen und ohne weitere Veränderung an der Benutzeranwendung vornehmbar. Abbildung 3.13 zeigt den Mechanismus. Wichtig ist hier das

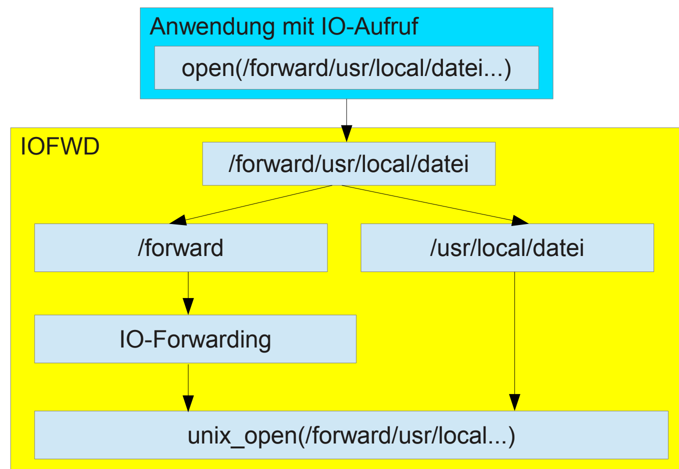

Abbildung 3.13: Spezieller Pfad für IO-Forwarding

der Pfad-Prefix auch wirklich auf dem Zielsystem vorhanden ist. Der Prefix wird nicht verworfen. Er wird im Zielsystem auf dem IOFWD Server tatsächlich verwendet.

3.7 Benutzung von IO-Funktionen

Als nächsten Schritt werden nun exemplarisch IO-Funktionen vorgestellt. Es ist zunächst zu bemerken, daß die IO-Funktion der Anwendung die IO-Funktion im IOFWD Client sind. Die Anwendung sieht keinen Unterschied zum originalen IO-Funktionsaufruf der zu der System C-Bibliothek geführt hätte. Es werden nun die einzelnen Funktionsblöcke in IOFWD Client und Server gezeigt, die die Funktionsaufrufe gruppieren. Die Anwendung springt direkt in IOFWD hinein mit ihrem ersten IO-Aufruf. Dieser Aufruf dient dazu, die Parameter zu überprüfen und für die nächste Aufrufstufe vorzubereiten. Beim open Aufruf werden bei der Rückgabe später auch die File Deskriptoren vermerkt und überprüft. Generell gibt es lokale File Deskriptoren und entfernte File Deskriptoren. Die

entfernten File Deskriptoren sind einfach die File Deskriptoren des IOFWD Servers plus eine Konstante. Abbildung 3.14 zeigt ein Beispiel für eine Konstante von 30000, die zu dem lokalen File Deskriptor hinzugefügt wird. Der IOFWD Server führt eine Liste mit

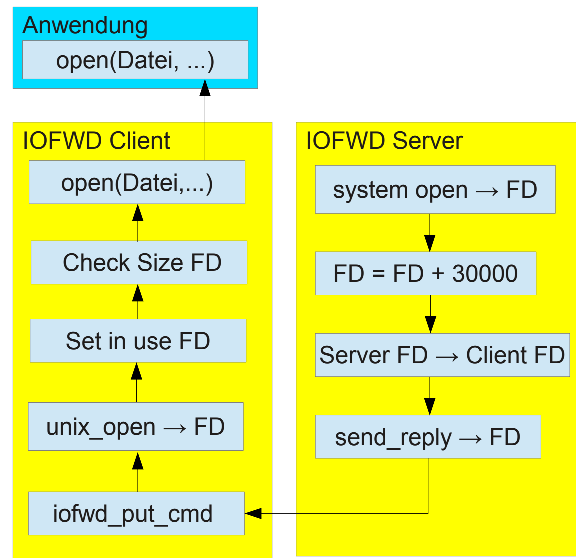

Abbildung 3.14: File Deskriptoren Übersetzung bei open Funktion

Client File Deskriptoren und zugehörigen Server File Deskriptoren. Auch der IOFWD Client führt ein Bit-Array der verwendeten File Deskriptoren. Die close Funktion nimmt den Client File Deskriptor und übersetzt diesen im Falle eines entfernten File Deskriptoren auf dem IOFWD Server in einen Server File Deskriptor (Abbildung 3.15 auf der nächsten Seite).

Die open Funktion der Anwendung wird auch abgeleitet zu IOFWD. Das bedeutet, beim Ablaufen der Anwendung wird das Symbol open erkannt und nach der entsprechenden Funktion in einer Bibliothek gesucht. Normalerweise wäre dies eine Standardbibliothek wie die glibc Bibliothek unter Linux. Doch anstelle tritt hier IOFWD mit seiner mit diesen den Symbolen zugehörigen Funktionen erstellten Bibliothek. Natürlich wird die eigentliche Funktionalität von open auch noch gebraucht. Dies tritt später auf der Server-Seite zum Vorschein. Man bezeichnet dieses eigentliche Funktion von open auch als „System“ open. Nach dem Eintreten in die open Funktion kann aufgrund der Pfadanalyse ein IO-Forwarding eintreten. Dazu wird der Inhalt der open Funktion in einen Puffer serialisiert. Dies ist eine bekannte Technik aus Java RMI oder CORBA, die eine leichte Übertragung von Parameterstrukturen ermöglicht. Unter Serialisierung versteht man das Verschicken von Datensätzen in seriell aneinandergereihten einzelnen Daten. Hierbei sind Marker dem Sender und dem Empfänger bekannt, d.h. die Methoden zum Entpacken und Packen der Daten. Die Methoden sind kongruent, sodaß die Daten auf der Empfängerseite wieder hergestellt werden können. Es gibt auch die Möglichkeit, automatisch

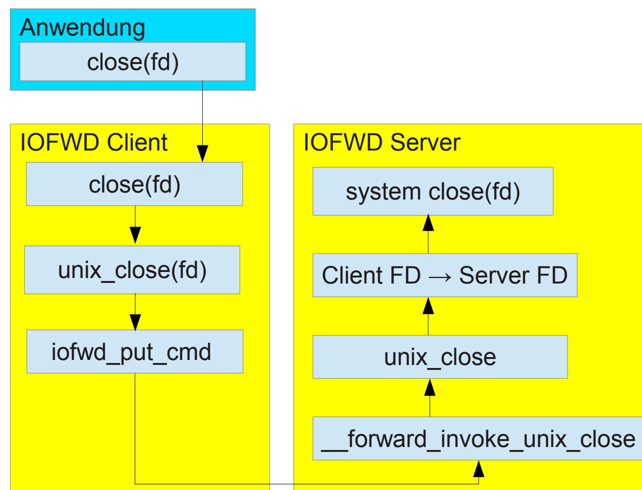

Abbildung 3.15: File Deskriptoren Übersetzung bei close Funktion

erkennen zu lassen, wie und wo die Daten liegen. Dies geht zu Lasten der Geschwindigkeit und ist bei der betrachteten Anordnung der Daten suboptimal. Der Aufwand zum Schreiben der „manuellen" Serialisierungs- und Deserialisierungsmethoden ist vernünftig. Automatismus wurde im Bereich der Funktionserzeugung betrieben. Abbildung 3.16 auf der nächsten Seite und 3.17 auf der nächsten Seite zeigen die komplette open Funktion mit Pfadanalyse. Die Deserialisierung und die Serialisierung wird nachfolgend genauer gezeigt. Hierbei wird vorgestellt, wie die Daten in aufeinander folgende Puffer gesetzt werden. Bei Serialisieren werden die Daten in diese Puffer hineingeschrieben. Dies geschieht beim Sender. Eine Besonderheit hier ist, daß nur Metadaten der Dateien auf diese Art übertragen werden. Der Übertrag der Dateiinhalte vom Sender zum Empfänger wird getrennt durchgeführt. Der Grund ist, daß Metadaten klein sind und schnell in einem Schritt übertragen werden können. Die Übertragung selbst muss daher nicht weiter optimiert werden. Bei der Übertragung der Dateiinhalte sieht das anders aus. Hier können große Datenmengen gesendet und empfangen werden. Eine Übertragung in einem Mal kann schon die Ressourcen der Knoten überschreiten. Da hier von Speicher zu Speicher übertragen wird, hat man Grenzen, die leicht bei den hier betrachteten Dateigrößen überschritten werden können. Daher werden die Dateiinhalte parallel in einem optimierten Prozess zur Übertragung großer Dateiinhalte stückweise und überlappend übertragen. Für die Übertragung der Dateiinhalte wird momentan server-seitig nur ein Thread verwendet. Client-seitig werden mehrere Threads eingesetzt, die zu mehreren Servern übertragen. Der Client verteilt die Arbeit auf die Server. Bei Erstellung von IOFWD wird festgelegt, auf wie viel Server der Client seine Aufrufe verteilt. Eine Zahl zwischen 4 und 8 Servern für den Client hat sich als vorteilhaft erwiesen. Der Server muss hier nicht speziell für diese Aufgabe instruiert werden. Dieser verrichtet seine Auf-

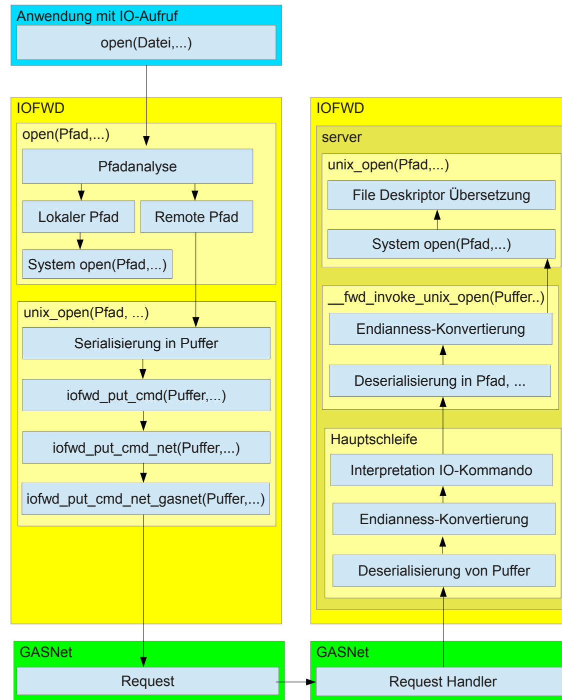

Abbildung 3.16: Kompletter Aufruf-Pfad von open Client nach Server

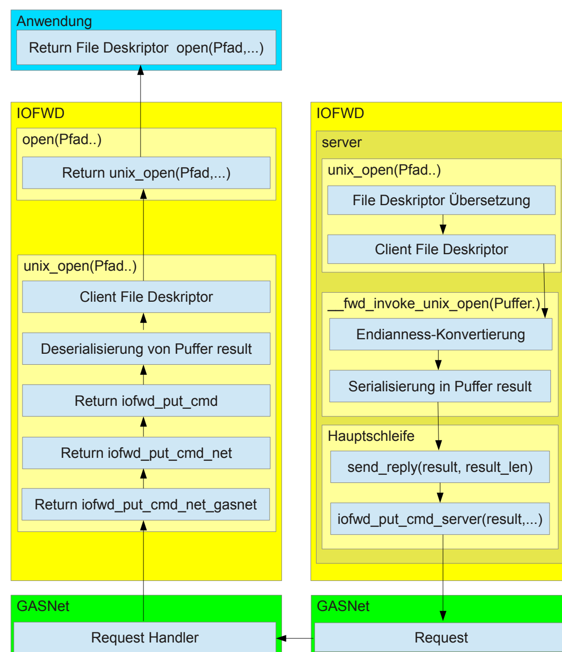

Abbildung 3.17: Kompletter Aufruf-Pfad von open Server nach Client

gabe genauso wie im - 1 zu 1 - Client zu Server Fall. Abbildung 3.18 und 3.19 zeigen die Serialisierung und Deserialisierung am Beispiel der read Funktion, die Dateiinhalte liest.

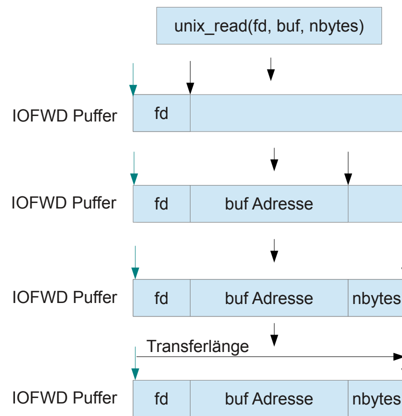

Abbildung 3.18: Serialisierung im Client am Beispiel read

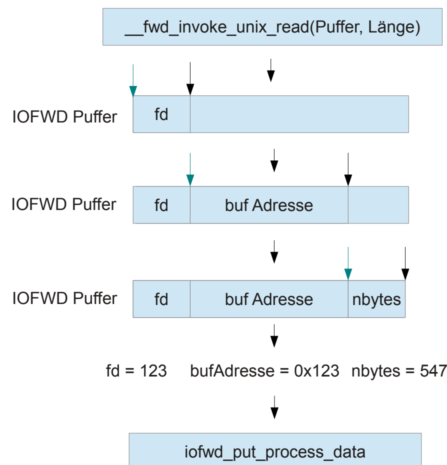

Abbildung 3.19: Deserialisierung im Server am Beispiel read

Prinzipiell werden die Kommando-Inhalte aus den IO-Aufrufen in einen Puffer sequentiell kopiert. Wichtig ist hier, daß hier völlig ohne Typen gearbeitet wird. Das bedeutet, es muß beim Deserialisieren bekannt sein, wie die Inhalte beim Serialisieren eingepackt wurden. Weiterhin wurde der read Aufruf als Beispiel ausgewählt, da hier der Puffer des Dateiinhaltes verwendet wird. Dies geschieht nur beim Lesen und Schreiben einer

Datei. Übermittelt im Serialisierungspuffer wird nur die Adresse des Puffers des Dateiinhaltes. Hier wird kein Dateiinhalt übertragen. Dies geschieht in einer zusätzlichen Aufruf-Sequenz (iofwd_put_and_process_data, iofwd_get_and_process_data).
Die Aufrufe read und write stellen Aufrufe dar, bei denen auch Dateiinhalte bewegt werden. Bevor man auf die Datei zugreifen kann, muss diese vorher mit einem open über einen Dateideskriptor referenziert werden. Dann kann man über den Dateideskriptor über read oder write auf die Datei und deren Inhalt über einen vorher bestimmten Puffer zugreifen. Die Puffer dieser Funktionen müssen mit den Puffern von IOFWD verbunden werden. Die Metainformation-Transferfunktionen (welche immer zuerst gestartet werden) stossen die Übertragung der Dateiinhalte an. Beim Erstellen von IOFWD mit dem Compiler und beigeordneten Skripten wird eine Datei gescannt, die nur Funktionsköpfe von Funktionen wie open, read und write mit kleinen Zusatzinformationen beinhaltet. Daraus generiert ein Skript die zugehörigen Funktionen für IOFWD mit Metainformationstransfer und Dateiinhaltstransfer auf Client- und Server-Seite. Es werden zunächst der Aufruf read (siehe Abbildung 3.20 und 3.21 auf der nächsten Seite) dargestellt. Dabei werden die verschiedenen Ebenen der Aufrufe gezeigt. Die Abstraktion wurde hier zu Vereinfachung weggelassen d.h. es gibt noch zusätzliche Aufrufebenen, die hier jedoch weniger zur Anschauung beitragen. Die Ebenen sind notwendig, um verschiedenartige Funktionalitäten, die nachfolgend erklärt werden, abzukapseln. Die Abstraktion, auf die auch später eingegangen wird, dient auch dazu, später leichter neue Transportnetzwerke in IOFWD einzubringen.

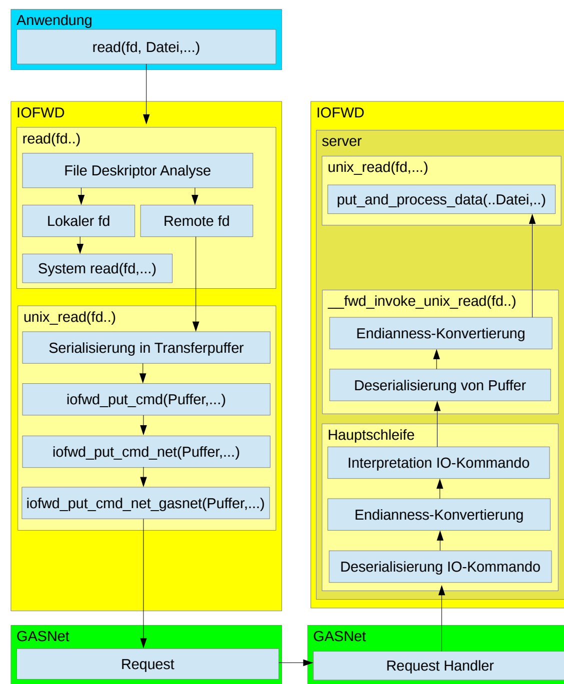

Abbildung 3.20: Aufrufkette read Client nach Server

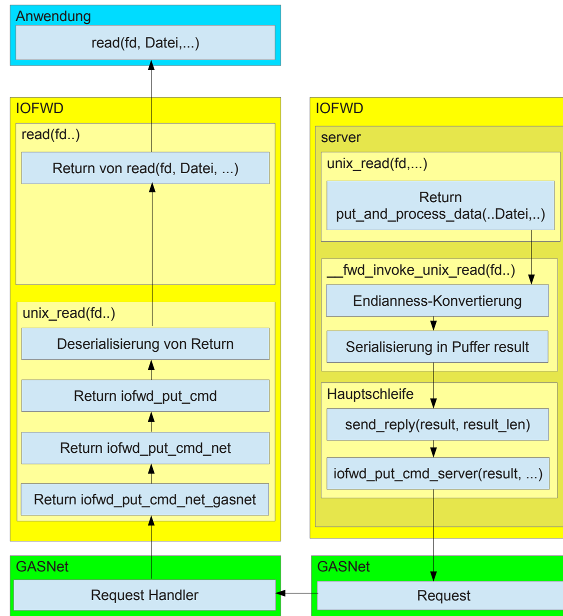

Abbildung 3.21: Aufrufkette read Server nach Client

Hier wird zunächst das read in der Anwendung zur read Implementation von IOFWD weitergeleitet. Es erfolgt eine Analyse, ob der File Deskriptor lokal - also nicht IOFWD - oder für IOFWD bestimmt ist. Man kann in Abbildung 3.20 auf der vorherigen Seite sehen, daß aufgrund der File Deskriptor Analyse ein entfernter File Deskriptor erkannt wird, der eine bestimmte Schwelle überschreitet (man erinnere sich im Beispiel war es 30000). Handelt es sich um einen lokalen File Deskriptor wird der Aufruf read zur Systembibliothek des Betriebssystems weitergeleitet. Da die Namen der Funktionen gleich sind, d.h. das open in IOFWD und und das open der Systembibliothek, hat sich IOFWD beim Start gemerkt wo sich die ursprünglichen Aufrufe befinden. Sonst wäre es schwer möglich diese Aufrufe in Systembibliothek des Betriebssystems wieder zu erreichen. Diese Fähigkeit ist auch wichtig, da auch beim Start von IOFWD, schon auf die IOFWD Funktionen eingesprungen wird, obwohl diese noch nicht voll funktionsfähig sind. Hier wird immer sofort auf die Systembibliothek verwiesen. iofwd_put_cmd hat den Sinn einer Kommandoweiterleitung des IOs an den Server. Hier wird lediglich mitgeteilt, wo der Dateiinhalt zu finden ist. Die Funktion put_and_process_data greift dies auf und erledigt den Transport der Dateiinhalte. Die Funktion iofwd_put_cmd teilt anhand einer Zahl mit, welches Kommando benutzt werden soll. In der Hauptschleife des Servers wird dies ausgewertet und aus einem Array von IO-Funktionen die richtige Funktion herausgenommen. Dies führt schließlich auf die unix_read Funktion des Servers. unix_read von Client und Server werden automatisch beim Erstellen von IOFWD erzeugt. Hierbei handelt es sich um die Funktions-Header der IO-Funktionen mit Kommentaren zu den Argumenten der Funktion. Daran erkennt ein Perl-Parser, wie er die entsprechenden IOFWD Funktionen aufzubauen hat.

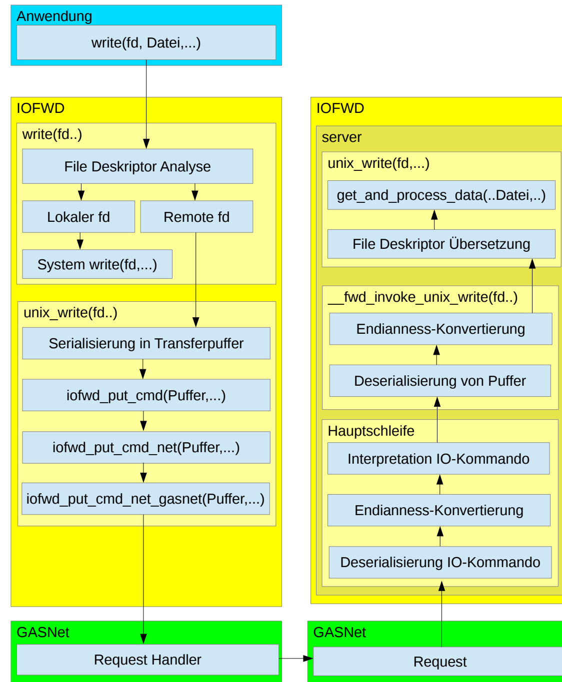

Abbildung 3.22: Aufrufkette write Client nach Server

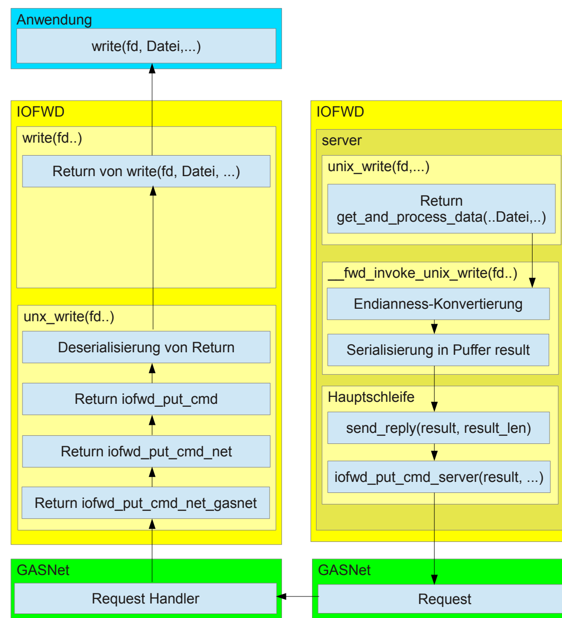

Abbildung 3.23: Aufrufkette write Server nach Client

Der Vorteil dieser Methode ist die einfache Erweiterung mit weiteren Funktionen. Funktionen mit Strukturen als Datensätzen werden in ihre einzelnen Datenelemente aufgeteilt. Der Parser muss diese Strukturen kennen. Das iofwd_put_cmd verschiebt einfach nur den Kommandopuffer. Die Intelligenz steckt in den Handler auf der Server-Seite. Der Handler empfängt den Puffer und läßt die Hauptroutine von einem Wartepunkt aus weiterlaufen. Dies geschieht, wenn der Handler das Kommando aus dem Kommandopuffer extrahiert hat und die entsprechende IO-Aufruf Routine startet. Man kann weiterhin sehen, daß eine spezielle Routine namens put_and_process_data aufgerufen wird, die sich um den Transport der Inhalte kümmert. Der write Befehl verwendet hier get_and_process_data. Der Server agiert als der aktive Part beim Transport der Dateiinhalte. Der Server schickt Dateiinhalte zum Client mit der put_and_process_data Sequenz im Falle von read oder fordert Dateiinhalte vom Client im Falle von write mit get_and_process_data an. Abbildungen 3.24 und 3.25 auf der nächsten Seite zeigen beide Operationen für read und write. Beide Operationen beinhalten das überschneidende

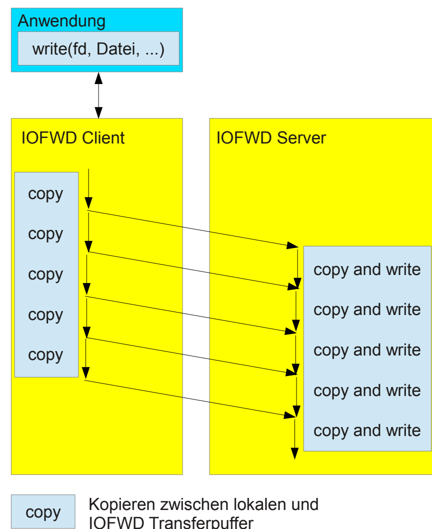

Abbildung 3.24: Bewegen von Dateiinhalten von IOFWD Server aufgrund von write

Kopieren von/in das Dateisystem und das Kopieren zwischen Client und Server. Es wird der gesamte Dateiinhalt stückweise bearbeitet und während zwischen Client und Server transferiert wird, wird schon das nächste Stück von/in das Dateisystem geschrieben. Auf Client- und auf Server-Seite wird auch zwischen lokalem Puffer und Transferpuffer von IOFWD kopiert. Ursprünglich war die Implementierung so, daß erst nach dem Transfer zwischen Client und Server wieder von/in das Dateisystem geschrieben wurde. Bei der Infiniband-Implementierung mußten noch zusätzliche Zwischenpuffer eingeführt werden, weil sich der IOFWD-Speicher nicht vollständig unter Infiniband registrieren lies.

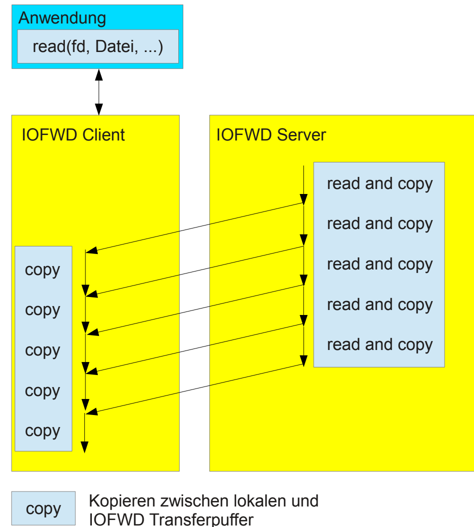

Abbildung 3.25: Bewegen von Dateiinhalten von IOFWD Server aufgrund von read

3.8 Threaded Umgebung

In der Threaded Umgebung ist es möglich, daß ein Client mit mehreren Server kommuniziert. Dabei wird über dasselbe Dateisystem ein und dieselbe Datei geöffnet, jedoch hat jeder Server seinen eigenen Bereich auf der Datei. Die Bereiche werden dann von den Threads parallel beschrieben. Abbildungen 3.26 auf der nächsten Seite und 3.27 auf der nächsten Seite zeigen die parallele Operation von einem Client auf drei Server. Zur Positionierung der Schreib- und Lesepunkte wird das lseek Kommando verwendet, daß auch eine Anpassung an eine wechselseitige Operation mit multiplen Hosts erfahren hat.
Prinzipiell wird eine Datei an allen drei Servern aufgemacht. Wird ein Inhalt geschrieben/gelesen wird dieser in drei gleichgroße Teile aufgeteilt und an der entsprechenden Position in der Datei geschrieben/gelesen. Eine Besonderheit ist hier das Schreiben ins „Leere“ ,d.h. die Datei muss nicht konsekutiv geschrieben werden. Man kann mit Löchern schreiben. Die Positionierung der Schreib-/Lesepunkte in nicht beschriebene Bereiche bereitet keine Probleme.
Der Kommandopuffer ist wie in Abbildung 3.28 auf Seite 48 aufgeteilt. Alle Funktionen, insbesondere die put/get Funktionen wurden um das gleichzeitige Benutzen des IOFWD Transferpuffers mit Kommandoinhalten erweitert. Im Wesentlichen wurden sie um die Adressierbarkeit von Knoten und Positionierung von Schreib-/Lesepunkten im IOFWD Transferpuffer erweitert. Abbildungen 3.29 auf Seite 48 und 3.30 auf Seite 49 zeigen ein Beispiel mit einem put_cmd, das für Knoten adressierbar ist. Die Aufspaltung in Threads hat auch eine Aufteilung der File Deskriptoren zur Folge. Diese Übersetzung ist momentan nur rudimentär gelöst. Es müßten jedoch Tabellen angelegt werden, welche eine Übersetzung von einem auf mehrere File Deskriptoren vorweisen können.

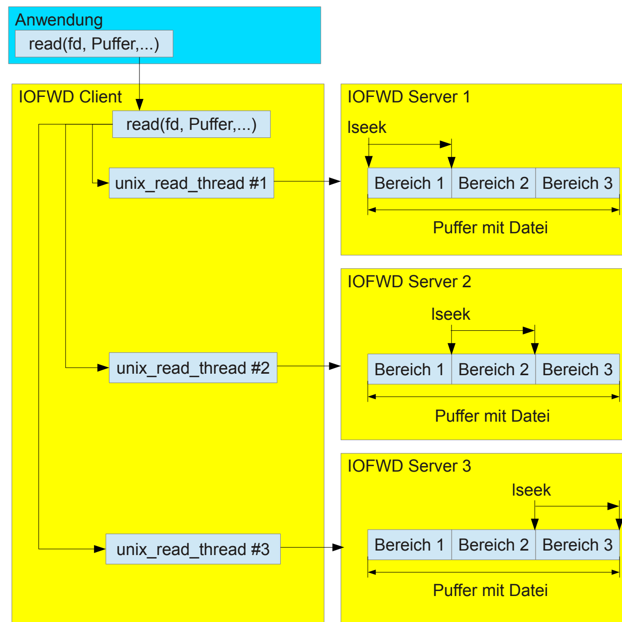

Abbildung 3.26: Ein Client operiert mit read mit mehreren Servern

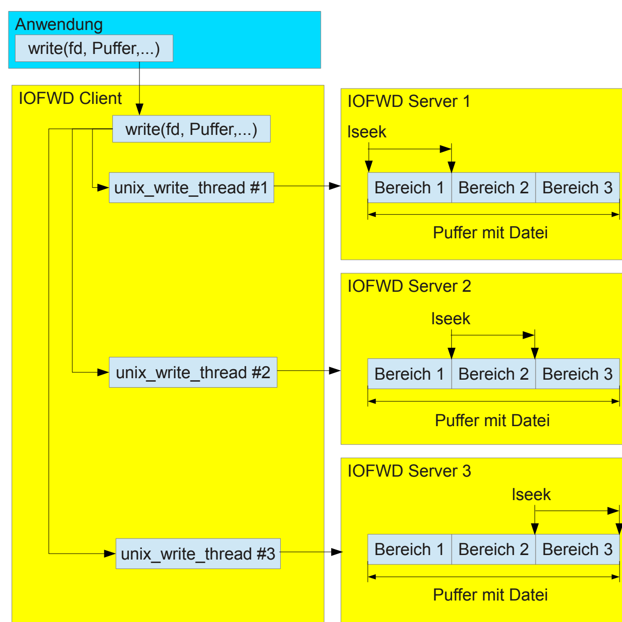

Abbildung 3.27: Ein Client operiert mit write mit mehreren Servern

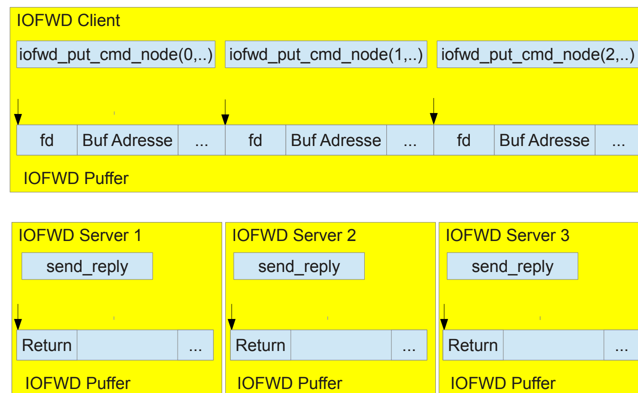

Abbildung 3.28: IOFWD Transferpuffer mit mehrfachen Kommandopuffern

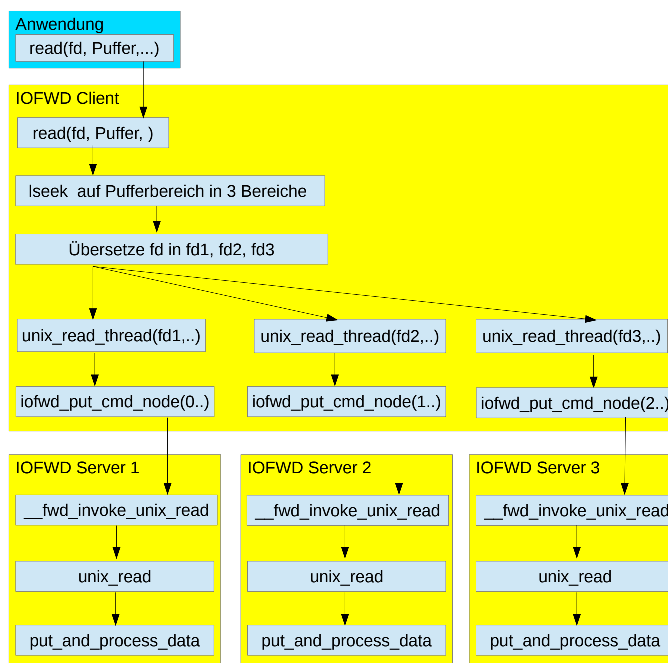

Abbildung 3.29: Ein Client mit 3 read Threads auf 3 Servern

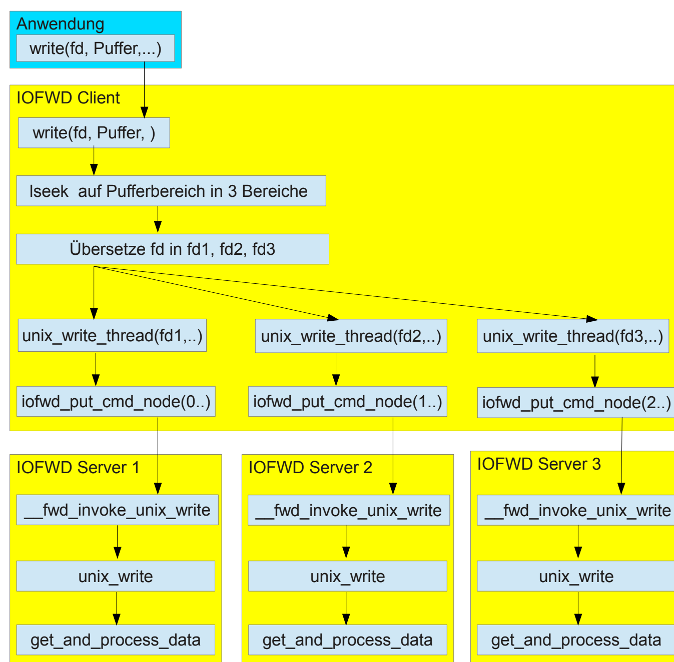

Abbildung 3.30: Ein Client mit 3 write Threads auf 3 Servern

3.9 Sammlung von IO-Aufrufen

Das Sammeln von kleinen IO-Aufrufen und anschließendes Auslösen von einem großen IO-Aufruf wird im Folgenden betrachtet. Viele kleine IO-Aufrufe durchzuführen, die Dateiinhalte transportieren, bedeutet auch eine erhebliche Menge an Metadaten. Die Verwaltungsoperationen werden pro kleinen Aufruf durchgeführt. Besser ist es, die Dateiinhalte zu sammeln und diese dann in einem einzigen Vorgang zu bewegen. Die gegenwärtig verwendete Methode in IOFWD sieht vor, daß der User die write/read Funktion mit einem vorangehenden Aufruf auf die Art und Menge der kleinen Schreib- bzw. Lesevorgänge vorbereitet. In dem Beispiel in Abbildung 3.31 auf der nächsten Seite gibt es drei kleine Schreibvorgänge. Der Benutzer gibt mit der write_prep Funktion einen Hinweis auf die absolute Länge aller Schreibvorgänge und deren Anzahl. Alle Schreibvorgänge müssen für diese Art der Optimierung gleich groß sein und konsekutiv geschrieben werden können, d.h. die kleinen Blöcken müssen aufeinander folgend im Quellspeicher liegen. Ein Unterschied zwischen Lesen und Schreiben ist, daß der Client beim Schreiben zwischenspeichert und der Server beim Lesen zwischenspeichert. Die Cachen der Daten erfolgt im Speicher, d.h. für diese Art der Optimierung muß der Speicher in Client und/oder Server entsprechend dimensioniert sein. Natürlich wird hier auch die Latenz erhöht. Das ist ein Nachteil dieses Verfahrens.

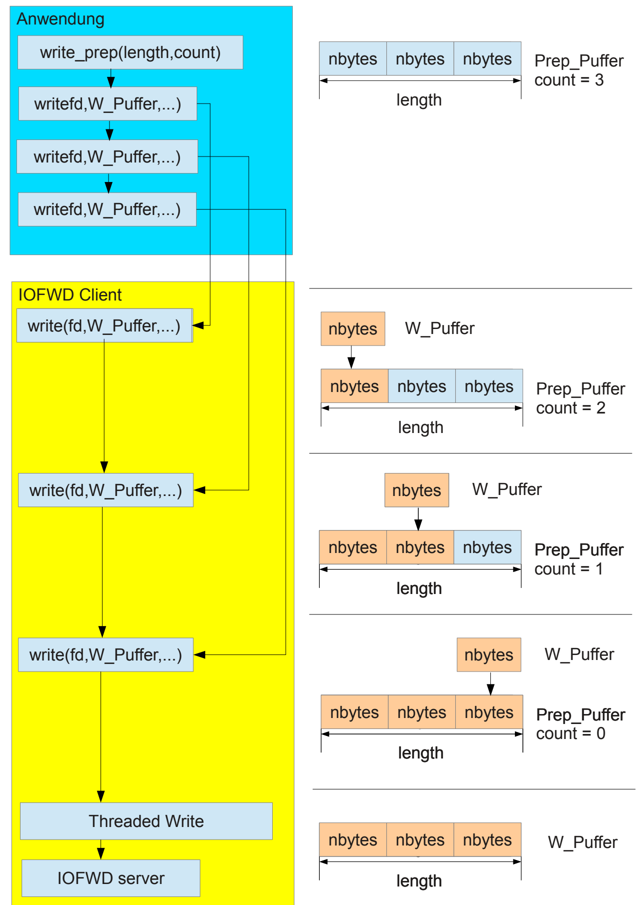

Abbildung 3.31: Sammeln von IO-Aufrufen

3.10 Abstrahierung

Die Funktionen in IOFWD waren zu Beginn zu spezifisch auf GASNet zugeschnitten. Es wurde eine „net“ Ebene in IOFWD eingeführt, die allgemein gültige Funktionen implementiert. Die Verallgemeinerung der Schnittstellen und Funktionen wurde mit GASNet und BMI erfolgreich getestet. Der Nachteil der Verallgemeinerung ist eine weitere Ebene der Indirektion, die sich jedoch nicht nachteilig ausgewirkt hat. Abbildung 3.32 auf der nächsten Seite zeigt die Ebenen der Abstraktion. Für den IOFWD Server gilt dasselbe Prinzip.

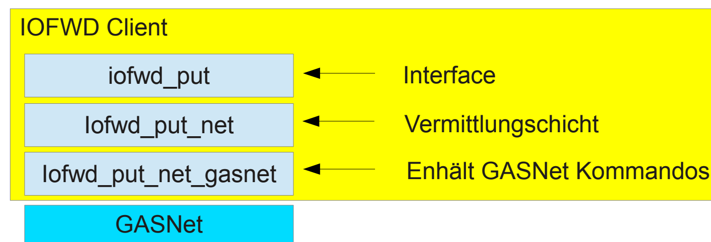

Abbildung 3.32: Abstraktionsebenen von IOFWD

4 Einbindung von IOFWD in eine Anwendung

4.1 IOFWD und Anwendung verbinden

Ohne IOFWD wird die Anwendung mit dafür vorgesehenen Skripten übersetzt (z.B. Makefiles). Dazu wird der Compiler aufgerufen und die Source-Dateien in Object-Dateien übersetzt. Die Object-Dateien fließen dann in Bibliotheken oder in die ausführbare Datei ein, mit der letztendlich die Anwendung gestartet wird.
Es gibt zwei Möglichkeiten IOFWD hier einzubringen:

- beim Erstellen der Anwendung
- bei der bereits erstellten Anwendung.

4.1.1 Anwendung wird erstellt

Beim Erstellen der Anwendung wird IOFWD mit einkompiliert. Dies ist immer der Fall, wenn die Anwendung statisch kompiliert wird. Die Hauptschleife der Anwendung muss eine Initialisierungssequenz für IOFWD in sich tragen, damit das Framework beim Ausführen der Anwendung gestartet wird. Alleinig die Initialisierungssequenz zieht die komplette Einbindung aller IOFWD Header Dateien und Bibliotheken wie auch der GASNet Header und Bibliotheken nach sich. Für GASNet werden dann die Bibliotheken für das entsprechende Netzwerk-Conduit eingebunden. Damit das Einbinden von IOFWD in die Anwendung während ihrer Erstellung möglichst einfach für den Anwender ist, wurde ein Compile-Wrapper in einer Skript-Sprache erstellt (Abbildung 4.1 auf der nächsten Seite).

Der Compile-Wrapper wird einfach anstelle des normalen Aufrufs für den Compiler verwendet. Im Compile-Wrapper wird die komplette Zeile eines Kompilier- oder/und Linkvorganges Element für Element abgefragt und ausgewertet. Es werden zwei Ziele verfolgt:

- Neubau des Kompilier- bzw. Kompilier- und Link Aufrufes mit IOFWD Bibliotheken und Header Dateien
- Einfügen der Wrap Vorganges.

Der Neubau des Aufrufes der Original-Compiler-Zeile geschieht im Preprocessing. Hier werden alle Elemente abgefragt und eingeordnet. Hier entscheidet auch der Zweck des

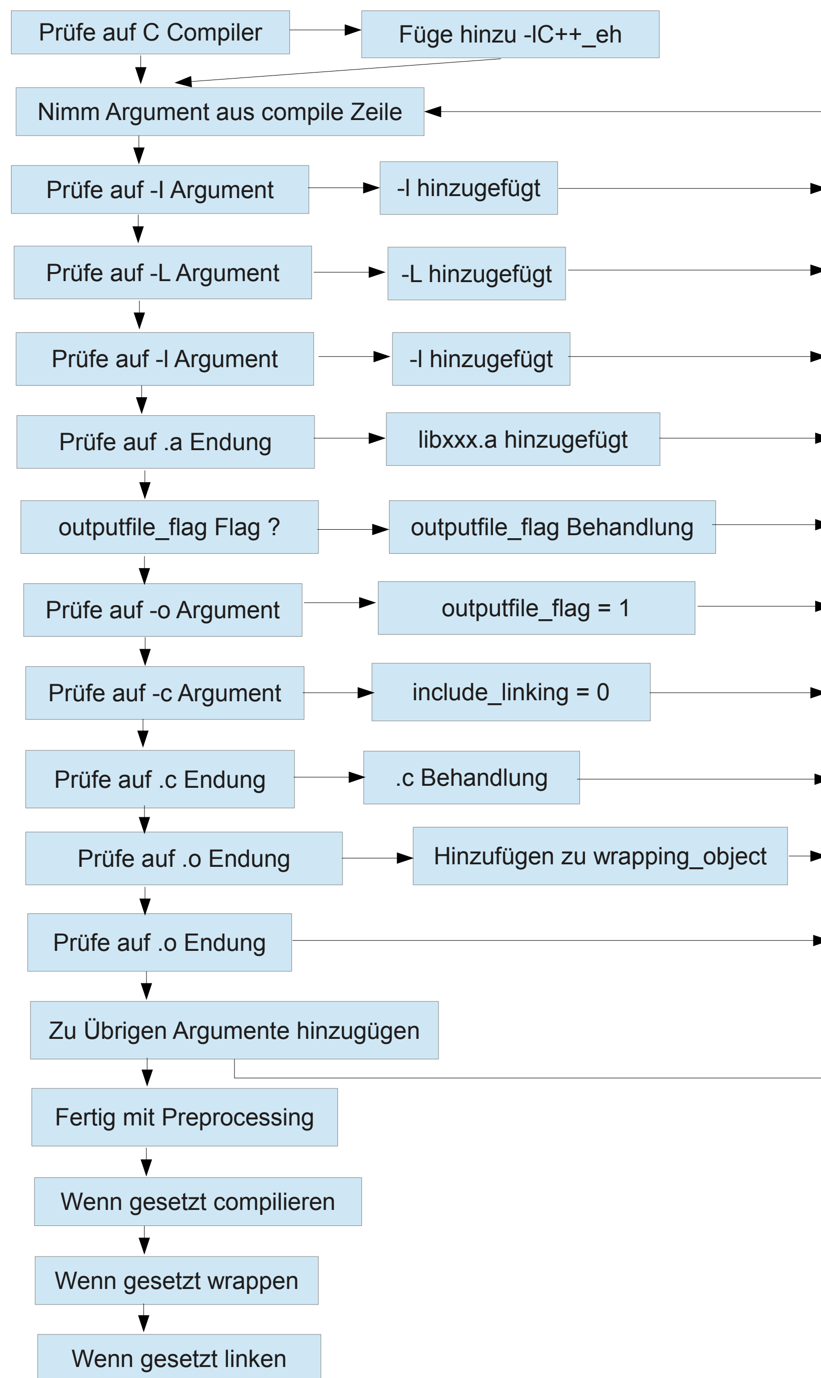

Abbildung 4.1: Compile-Wrapper Preprocessing

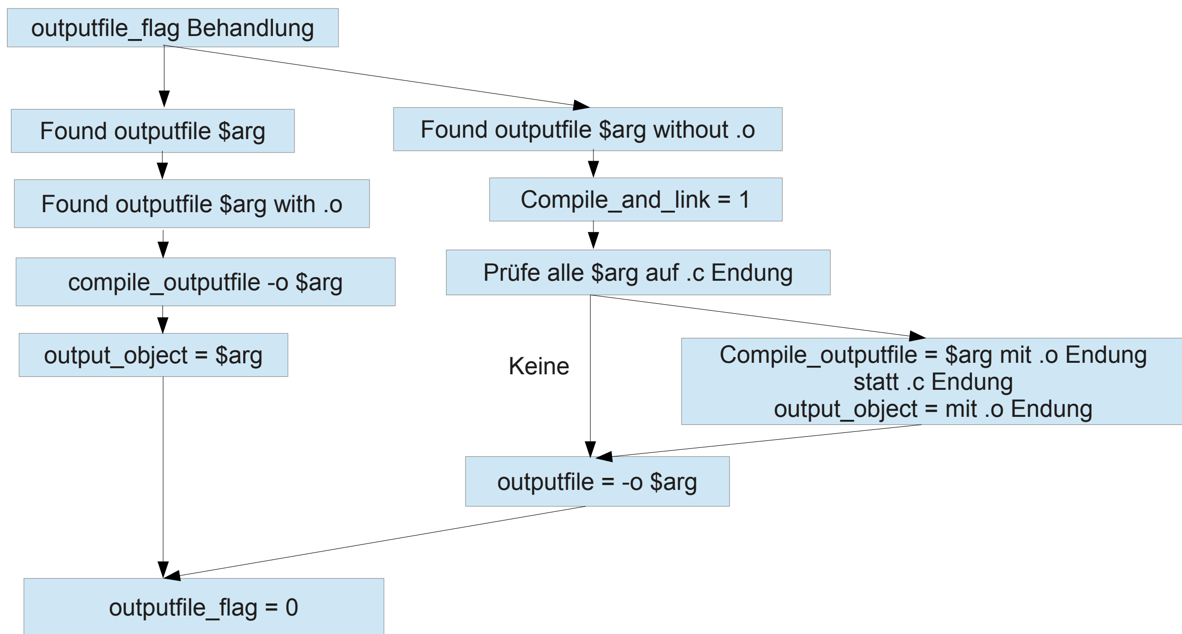

Abbildung 4.2: Compile-Wrapper outputfile Subroutine

ursprünglichen Aufbaus darüber, wie verfahren werden muß. Prinzipiell wird nur kompiliert, nur gelinkt oder kompiliert und gelinkt. Beim nur kompilieren wird das entstandene Objekt anschließend gewrappt, d.h. die Symbole der IO-Aufrufe werden gegen __wrap Symbole ausgetauscht. Diese führen bei einem IO-Aufruf zu IOFWD. Wird in einem Schritt kompiliert und gelinkt, muss dieses aufgetrennt werden, damit gewrappt werden kann. Das Wrappen kann nur an Object-Dateien erfolgen. Daher wird das Ein-Schritt Kompilieren und Linken in ein Drei-Schritt Kompilieren, Wrappen und Linken aufgeteilt. Zum Schluss wird automatisch die IOFWD Library beim Linken hin zugebunden. Compile-Wrapper, Bibliotheken und Header für IOFWD und GASNet sind im System vorinstalliert. Der Name des Compile-Wrappers reicht als Information für den Ersteller der Anwendung aus. Der Compile-Wrapper kennt die Lokation der benötigten Bibliotheken und Header-Dateien für IOFWD und GASNet. Der Compile-Wrapper ist für

- C
- C++
- Fortran

verfügbar. Eine Besonderheit auf der SX-Platform ist die Verwendung eines portierten GNU-Linkers. Der Linker kann Symbole wrappen, d.h. er kann Aufrufe durch andere Aufrufe in Object-Dateien ersetzen. Als Beispiel der Aufruf open: dieser wird durch

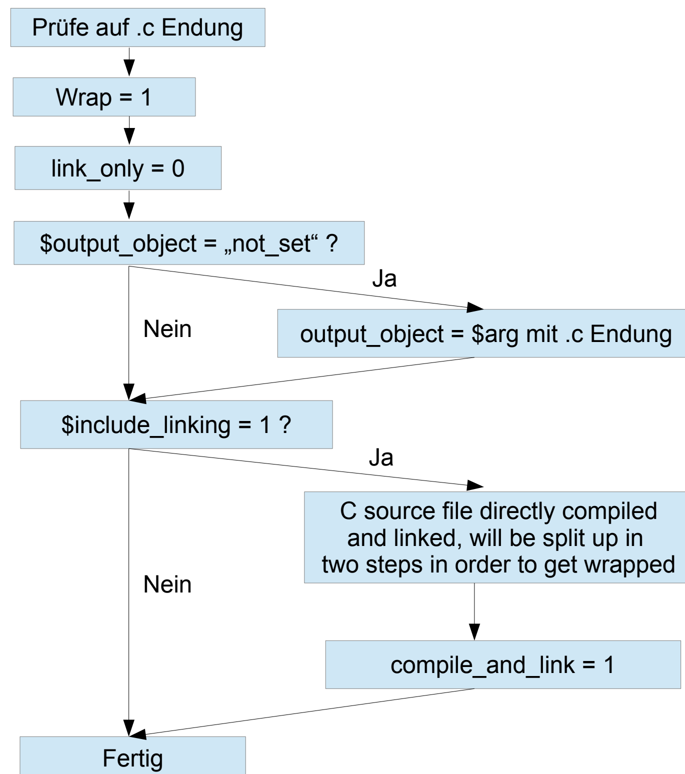

Abbildung 4.3: Compile-Wrapper cending Subroutine

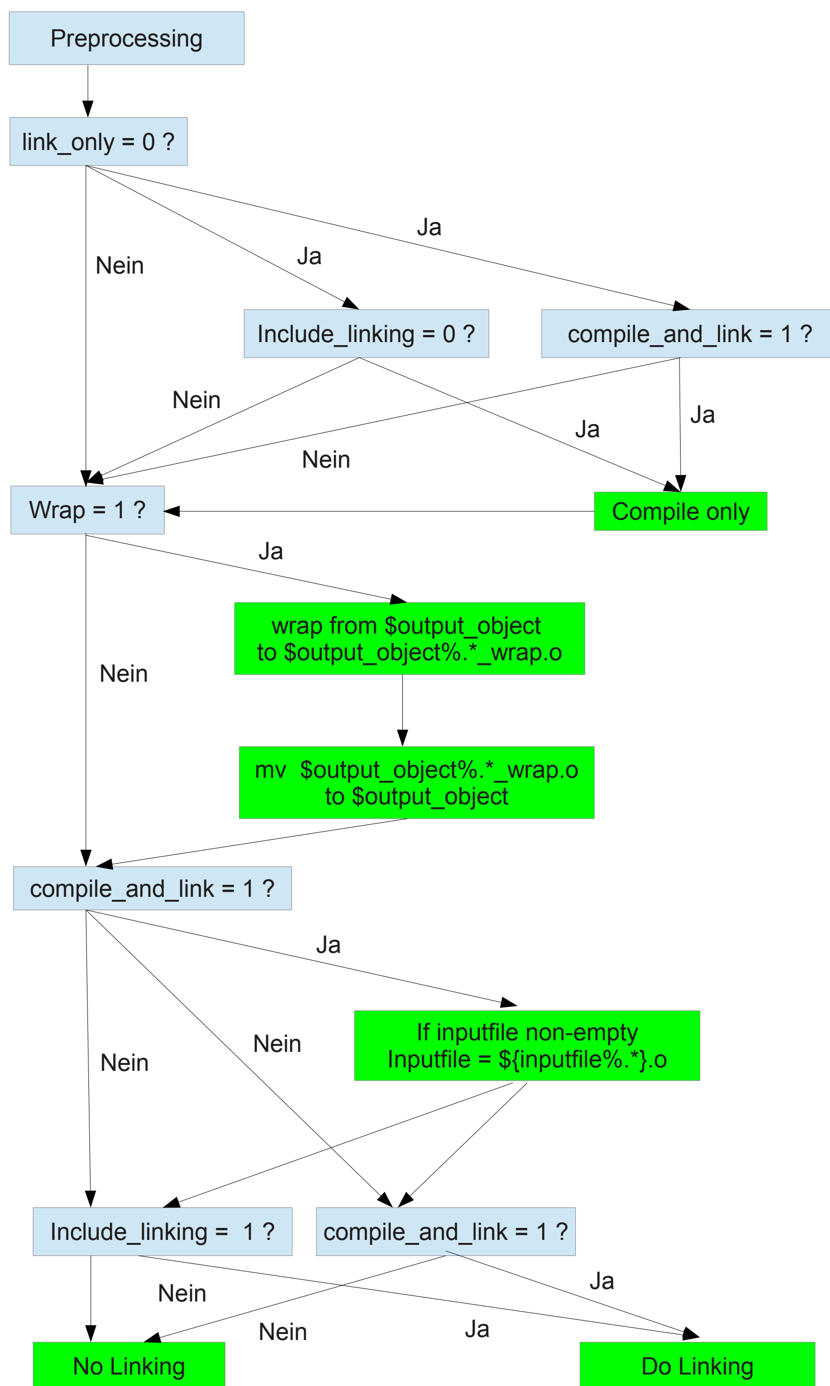

Abbildung 4.4: Compile-Wrapper Wrap, Compile und Link Prozedur

__wrap_open ersetzt, was direkt zu IOFWD führt. Der Compile-Wrapper benutzt den GNU-Linker zum wrappen von Symbolen und den SX-Linker zum Einbinden der IOFWD Bibliothek. Der Compile-Wrapper analysiert, wie die ausführbare Datei der Anwendung entsteht, und wrapped und bindet dementsprechend die Object-Dateien.
Diese Vorgehensweise ist bei C/C++ Programmen gut durchführbar. Schwierigkeiten bereitete das IO-Forwarding für Fortran-Programme. Der Grund hierfür ist das Fortran eigene IO-Operationen verwendet, die zu einem späteren Zeitpunkt die eigentlichen IO-Aufrufe, die IOFWD abfängt, verwenden. Daher wurden Wrapper-Skripte erstellt, die während der Installation von IOFWD auf dem System, die Fortran System-Bibliotheken analysieren und die eigentlichen IO-Operationen in den Fortran IO-Operationen finden und diese dann ersetzten. Dabei gibt es auch in der C-System-Bibliothek auf der SX Platform Aufrufe, die die von IOFWD unterstützten IO-Aufrufe verwenden. Daher wird auch zusätzlich die C-System-Bibliothek auf der SX-Platform während der Installation von IOFWD analysiert und entsprechende Symbole ersetzt. Dies hat zur Folge, dass neben den 43 allgemein unterstützten IO-Operationen noch zusätzlich Stream Funktionen unterstützt werden. Getestet wurden davon lediglich folgende Funktionen:

- fopen
- fread
- fwrite
- fclose.

Der Compile-Wrapper für dynamisch erzeugte Binaries funktioniert auf die dieselbe Art und Weise. Der Unterschied ist das eine dynamisch ladbare Version von GASNet und IOFWD Libraries verwendet werden. In diesem Fall reicht es aus, wenn die Object-Dateien der Anwendung vorhanden sind. Diese können gewrapped und mit IOFWD gebunden werden.

4.1.2 Erstellte Anwendung

Im Falle einer erstellten Anwendung gibt es folgende Ausgangssituationen:

- die Anwendung ist statisch: umfangreich, hier wird ein Weg skizziert mit Instrumentierung (Präparieren der Anwendung, siehe nachfolgenden Abschnitt)
- die Anwendung ist dynamisch
  - IOFWD Preload: nur mit Instrumentierung
  - IOFWD Full Preload: mit und ohne Instrumentierung, je nach Preload-Kapazität der Laufzeitumgebung.

Wenn der Preload Mechanismus der Laufzeitumgebung greift, ist die Full Preload-Lösung direkt einsetzbar. Funktioniert der Preload Mechanismus nicht, muß man die Anwendung instrumentieren. In jedem Fall muß man die IOFWD Preload Lösung instrumentieren.

Instrumentierung

Bei der Instrumentierung wird die Anwendung von einem Framework bearbeitet, das die Anwendung im Speicher [26] oder in der Datei verändert und ein Laden von IOFWD ermöglicht. Man nennt das Verändern der ausführbaren Datei statische und das Verändern der Anwendung im Speicher dynamische Instrumentation.
Es existieren Frameworks wie EEL [27], BIRD [28], Pin [29], PEBIL [30], die eine Veränderungen der Anwendung erleichtern. Nach kurzem Studium dieser Frameworks ist die Entscheidung für DynInst [31] gefallen. Das DynInst API ist gut zu bedienen, es wird aktuell weiterentwickelt und ist gut verständlich aufgebaut.
Die DynInst Library ist maschinenunabhängig konzipiert, patcht aber den Code auf Maschinenebene. Hier ist kein Neukompilieren oder Re-Linking des Codes erforderlich. Es existiert ein objekt-basiertes Interface, das es ermöglicht neuen Code einzufügen (Abstract Syntax Trees - AST).
Die grundlegenden Operationen von DynInst sind unter anderem

- Code-Abfrageroutinen
- Code-Modfikationsroutinen
- Erzeugung von AST Code-Sequenzen.

Die Code-Modifikationsroutinen sind unter anderem

- Entfernung von Funktionsaufrufen
- Ersetzung von Funktionsaufrufen
- Ersetzung von Funktionen
- Wrappen von Funktionen.

Weiterhin gibt es noch Operationen wie

- malloc/free - Allozieren von Heap Space im Applikationsprozess
- Asynchrones Ausführen einer Funktion in der Applikation
- Laden einer Shared Library in die Applikation.

Das DyninstAPI bietet weitere APIs in weiteren Schichten:

- SymtabAPI [32]
- StackwalkerAPI [33]
- InstructionAPI [34]
- DepGraphAPI [35].

Diese APIs bauen teilweise aufeinander auf und können auch direkt benutzt werden. Das SymtabAPI ist ein Tool zum Parsen von Symbol Tabellen, Object Headers und Debug Informationen. Es unterstützt die Formate ELF, XCOFF und PE. Es unterstützt auch die DWARF- und Stabs Debugging-Formate. Das Hauptziel dieses APIs ist es, eine abstrakte Sicht auf Binaries und Libraries zu werfen. Dies gewährt Plattform-Unabhängigkeit. Das StackwalkerAPI ist eine Library, die es ermöglicht, den Call Stack zu durchschreiten. Den Call Stack finden man in einem Prozess, der die geraden aktiven Stack Frames beinhaltet. Jeder Stack Frame beinhaltet Daten von einer ausführenden Funktion. Das InstructionAPI übersetzt zwischen der Darstellung in dem Binary und einer abstrakten Form. Diese abstrakte Form ist nötig, um Binaries besser modifizieren und analysieren zu können und so auch eine Plattform-Unabhängigkeit für die übergeordneten Funktionen zu gewährleisten. Grundlegend hat das InstructionAPI die Eigenschaften Binary Code zu dekodieren, die Darstellung von Maschineninstruktionen zu abstrahieren und Code zu dissassemblieren. Das DepGraphAPI erzeugt einen Abhängigkeitsgraphen vom binären Code. Hierbei repräsentieren Knoten im Graphen Programmelemente und die Kanten Abhängigkeiten, die Aktionen zwischen den Programmelementen beschreiben. Es gibt vier unterschiedliche Graphentypen:

- Data Dependence Graph (DDG)
- Control Dependence Graph (CDG)
- Program Dependence Graph (PDG)
- Extended Program Dependence Graph (xPDG).

IOFWD Full Preload Lösung

Der einfachste Fall von allen ist die funktionierende Preload Fähigkeit der Laufzeitumgebung für die jeweilige Anwendung d.h. die Anwendung erlaubt es, einen Preload durchzuführen. Dann wird die IOFWD Bibliothek vor allen anderen von der Anwendung nach Funktionen durchsucht (Abbildung 4.5 auf der nächsten Seite).
Läßt die Anwendung ein Laden über den Preload Mechanismus nicht zu, so kommt die Lösung mit einer Instrumentierung zum Tragen. Dabei wird über die Instrumentierung ein Eintrag in der Anwendung vorgenommen, daß die IOFWD Bibliothek von der Anwendung als erstes geladen und nach Funktionen durchsucht werden soll. Hierzu wurden die Quellen von DynInst geändert, weil ein Eintrag der IOFWD Bibliothek in die Anwendung vorher nicht möglich war. Dazu wurden die sogenannten DT_NEEDED Records so ergänzt, daß die IOFWD Bibliothek als erstes eingetragen wird. Die Änderung wurde im SymtabAPI von DynInst durchgeführt. Dies geschieht automatisch beim Einlesen und anschließenden Neuschreiben des ausführbaren Binaries. Folgendes Beispiel zeigt das Hinzufügen von der IOFWD Bibliothek in das Binary a.rewritten.out:

```
BPatch bpatch;
BPatch_binaryEdit *app_bin = NULL;
app_bin = bpatch.openBinary("a.out", true);
```

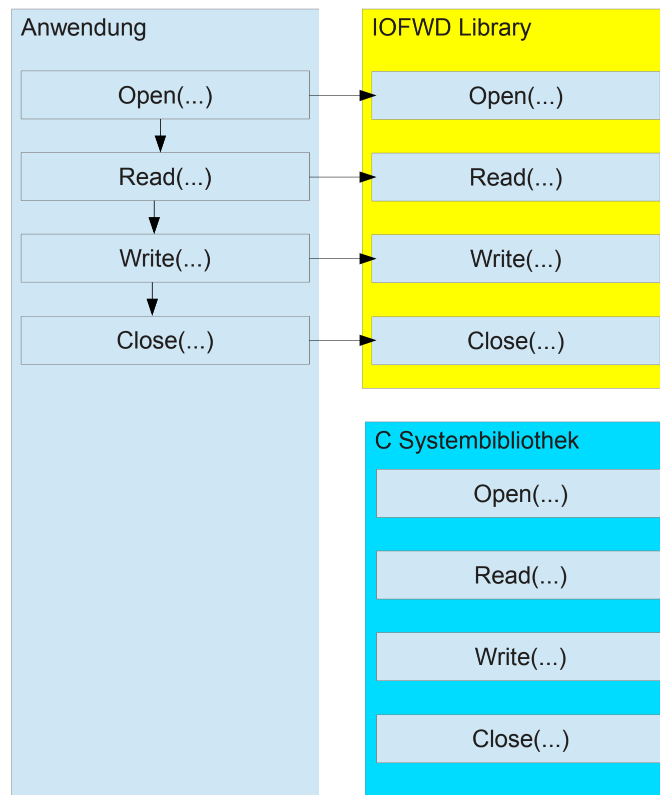

Abbildung 4.5: Aufruf-Präferenz

```
app_bin->writeFile("a.rewritten.out");
```

Die Library Abhängigkeiten der Anwendung vor der Veränderung:

```
libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x0000003e74600000)
libm.so.6 => /lib64/libm.so.6 (0x0000003821400000)
libc.so.6 => /lib64/libc.so.6 (0x0000003821000000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x0000003e74200000)
/lib64/ld-linux-x86-64.so.2 (0x0000003820000000)
```

und nach dem Hinzufügen der IOFWD Library:

```
libiofwd.so => ./libiofwd.so (0x00002...)
libc.so.6 => /lib64/libc.so.6 (0x0000003821000000)
libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x0000003e74600000)
libm.so.6 => /lib64/libm.so.6 (0x0000003821400000)
libdl.so.2 => /lib64/libdl.so.2 (0x0000003821800000)
/lib64/ld-linux-x86-64.so.2 (0x0000003820000000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x0000003e74200000)
```

IOFWD Preload Lösung

Bei der Preload Lösung von IOFWD gibt es auch die zwei Fälle, ob ein Preload von der Laufzeitumgebung funktioniert oder nicht. Dann muß die Anwendung auch wie in der Full Preload Lösung mit einem Bibliothekseintrag in der Anwendung instrumentiert werden. In jedem Fall müssen die IO-Funktionen durch __wrap IO-Funktionen ausgetauscht werden. Die __wrap Funktionen stehen in diesem Fall in der IOFWD Bibliothek. Der Eintrag der IOFWD Bibliothek wird in diesem Fall auch automatisch in der Anwendung vorgenommen, da DynInst Kenntnis davon hat, dass Symbole aus der Bibliothek IOFWD verwendet werden. In jedem Fall werden die Aufrufe für die IO-Funktionen wie an dem Beispiel für open gezeigt, durchgeführt:

```
BPatch bpatch;
bool res;
BPatch_addressSpace *addr_space = NULL;
BPatch_binaryEdit *app_bin = NULL;
app_bin = bpatch.openBinary("a.out", true);
addr_space = static_cast<BPatch_addressSpace *>(app_bin);
res = addr_space->loadLibrary("libiofwd.so",true);
if ( res == true )
        printf("library loaded ok.\n");
else
        printf("library NOT loaded ok.\n");
BPatch_image *appImage = addr_space->getImage();
BPatch_Vector<BPatch_function *> functions,functions2;
appImage->findFunction("__wrap_open", functions);
```

```
appImage->findFunction("open", functions2);
addr_space->replaceFunction(*functions2[0],*functions[0]);
app_bin->writeFile("a.rewritten.out");
```

Hierin wird das ausführbare Programm a.out als Datei zunächst geladen. Danach wird die Library libiofwd.so hinzu geladen. Diese Library enthält die Funktion __wrap_open, die die Funktion open im Hauptprogramm von a.out ersetzen soll. Es werden danach die Funktionen open und __wrap_open gesucht. In replaceFunction werden alle Funktionsaufrufe von open mit __wrap_open ersetzt. Zum Abschluss wird das veränderte Binary wieder in die Datei a.rewritten.out zurückgeschrieben. Die Library-Abhängigkeiten vor der Änderung:

```
libc.so.6 => /lib64/libc.so.6 (0x0000003821000000)
/lib64/ld-linux-x86-64.so.2 (0x0000003820000000)
```

Und nach der Änderung:

```
libdyninstAPI_RT.so.1 => ./libdyninstAPI_RT.so.1 (0x00002...)
libiofwd.so => ./libiofwd.so (0x00002...)
libc.so.6 => /lib64/libc.so.6 (0x0000003821000000)
libdl.so.2 => /lib64/libdl.so.2 (0x0000003821800000)
/lib64/ld-linux-x86-64.so.2 (0x0000003820000000)
```

Man sieht, daß das Binary a.rewritten.out jetzt die Library libiofwd.so verwendet. Außerdem bemerkt man, daß eine Run-Time Library von DynInst geladen wird. Beim Ausführen des neu erzeugten Binaries a.rewritten.out wird nun anstatt der Funktion open die Funktion __wrap_open aus der Library libiofwd.so aufgerufen. Das Symbol für die Funktion __wrap_open taucht im Binary a.rewritten.out nicht auf. Es bleibt das alte Symbol open.

Statische Anwendung

Es gibt keinen Mechanismus in der Laufzeitumgebung, der es ermöglicht, Code von IOFWD zu der statischen Anwendung hinzuzufügen. Über die Instrumentierung ist das Einbringen von Code von IOFWD in die Anwendung möglich. Der Unterschied ist hier, daß beim dynamischen Fall Bibliothekseinträge und gegebenenfalls Funktionsaufrufe in der Anwendung abgeändert werden. Beim statischen Fall muß zusätzlich Code von Funktionsinhalten in die Anwendung eingebracht werden.
Das bedeutet, daß der C-Code, in dem IOFWD geschrieben ist, in die Sprache für DynInst umgewandelt werden muß und dann eingebracht werden kann. So könnten komplett alle IOFWD Funktionen in das statische Binary gelangen. Das kleine Code-Beispiel in C zeigt, wie hier bei DynInst vorgegangen wird.

```
while (i < 3) {
        i++;
}
```

Der Code wird in folgendes Konstrukt für DynInst umgewandelt:

```
BPatch_nullExpr loopDone;
// if (i > 3) goto loopDone:
// 1. Boolscher Ausdruck (i > 3)
BPatch_boolExpr testFlag(BPatch_gt, *intI, BPatch_constExpr(3));
// 2. if Ausdruck und goto Ausdruck
BPatch_ifExpr test(testFlag, BPatch_gotoExpr(loopDone));
//
// i++
BPatch_arithExpr addOne(BPatch_assign, *intI,
       BPatch_arithExpr(BPatch_plus, *intI, BPatch_constExpr(1)));
// Zusammenfassung der vorherigen Statements in ein einziges
// Statement
BPatch_Vector<BPatch_snippet *> statements;
statements.push_back(&test);
statements.push_back(&addOne);
statements.push_back(&loopDone);
// Einziges Statement
BPatch_sequence whileLoop(statements);
```

Am Ende der Instruktionen werden die sogenannten „Snippets“, also Code-Segmente zu einer Sequenz zusammengefasst, die dann an einem definierten Punkt im statischen Binary eingesetzt werden. Die Snippets können auch Funktionen sein. Für diese Lösung müßte ein Parser entworfen werden, der Code automatisch in die DynInst-Sprache umwandeln kann, damit diese Aufgabe in absehbarer Zeit zu bewältigen ist.
Zukünftige Pläne für DynInst sind es, eine Library in ein statisches Binary zu laden. Mögliche Ideen hierzu sind die Library am Ende des statischen Binaries einzufügen und DynInst die Referenzen auflösen zu lassen. Dies ist jedoch problematisch, wenn das statische Binary gestripped ist. Auch existiert bisher keine Infrastruktur im statischen Binary, die es ermöglicht, Libraries hinzu zu laden.

4.2 Zugriff auf Dateisysteme ohne Dateisystem-Client

Da IO-Forwarding sich um die Abwicklung der eigentlichen Dateioperationen auf dem Dateisystem kümmert, ist nur ein minimaler Dateisystem-Client auf dem Rechenknoten notwendig. Dieser stellt das lokale Dateisystem für den Rechenknoten, idealerweise im Speicher, zur Verfügung. Es besteht auch die Möglichkeit einen minimalen IO-Forwarding Dateisystem Client z.B. mit FUSE [15] anzulegen. FUSE ist ein generisches Dateisystem im User Space, das es einfach ermöglicht, ein spezifisches Dateisystem zu implementieren. Es existiert auch eine FUSE Implementation von IOFWD. Hierbei ist darauf zu achten, daß in diesem Fall Dateizugriffe gecacht werden. Um ein Cachen zu

vermeiden sollte die Datei und das übergeordnete Directory lokal mit einem fsync-Aufruf im FUSE-Dateisystem nach der Lese- bzw. Schreiboperation versehen werden. Dies sorgt dafür, daß der Cache auch tatsächlich von IOFWD übertragen wird.
Somit haben die Rechenknoten Zugriff auf eine Vielzahl von Dateisystemen, ohne dafür einen Dateisystem-Client zu benötigen. Im Falle des NEC SX-Systems existiert keine Unterstützung für das Lustre-Dateisystem. Mit IO-Forwarding ist es möglich, Daten mit dem Lustre Dateisystem auszutauschen.
Die Senkung von Kosten ist z.B. durch die Verwendung eines neuen Rechenclusters mit bereits existierenden Datenvorrichtungen mit IO-Forwarding möglich.

5 Native Anwendung

Während der Entwicklung von IOFWD mußten verschiedene funktionelle Bestandteile des Systems wiederholt gestestet werden. Auch um bestimmte Funktionalität wie den Aufruf der verschiedenen IO-Operationen wie open, read, lseek u.s.w. oder Fortran-Programme zu testen. Auch ein einfaches MPI-Programm hat seinen Weg in die Test-Suite gefunden. Die nativen Anwendungen sind vornehmlich einige Test-Anwendungen aus IOFWD. Vor allem stellt diese Bezeichnung hier Anwendungen dar, die in keiner speziellen Umgebung wie PGAS oder MPI laufen. Die Anwendung, die hier gestestet wird, heißt nbmrw und gibt den Datendurchsatz für Lesen und Schreiben für verschiedene Dateigrößen aus. Die Größenvorgaben für die Dateien sind einstellbar in einem von bis Intervall und verdoppeln sich in jedem Schritt.

5.1 Erstellung der Anwendung

Die native Anwendung nbmrw aus der Test-Suite von IOFWD wurde mit dem Compile Wrapper für C übersetzt. Die Anwendung nbmrw wurde dynamisch erzeugt. Es wurde die Preload-Lösung verwendet.

5.2 Test-Aufbau

Im Infiniband-Netzwerk des Linux Clusters wurde der Aufbau wie Abbildung 5.1 auf der nächsten Seite getestet. Für die Messungen ohne IO-Forward wurde einfach direkt lokal in das Lustre-Dateisystem eingeschrieben und ausgelesen. In beiden Fällen mit und ohne IOFWD wurde mit 8 OSTs in Lustre gestriped. Für die IOFWD-Messungen wurden die Daten über das Infiniband-Netzwerk an die IOFWD Server und dann wieder über Infiniband an die Plattensysteme übertragen.

5.3 Durchführung der Tests

Die Abbildungen 5.2 auf Seite 67, 5.3 auf Seite 68, 5.4 auf Seite 69 und 5.5 auf Seite 69 zeigen die Performance über natives Infiniband mit und ohne IOFWD. Es wurde hierbei ein Box-and-Whisker-Diagramm benutzt, das folgende Elemente enthält:

- kleinster Wert
- unteres Quartil (Viertelwert)

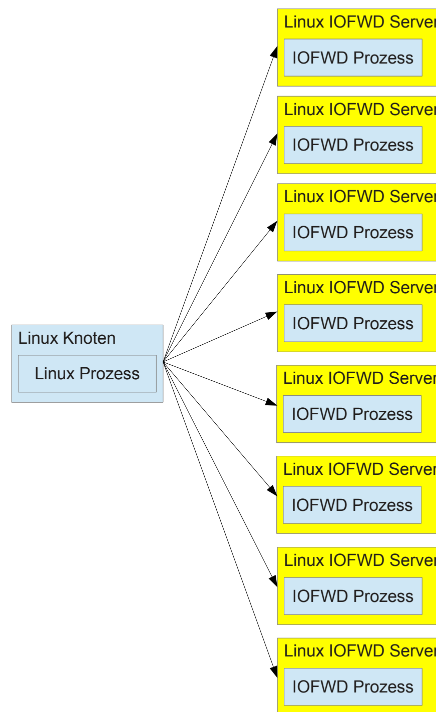

Abbildung 5.1: Anwendung nbmrw mit Threaded IOFWD Client

- Median (Mittelwert)
- oberes Quartil
- maximaler Wert
- Ausreisser.

Diese Darstellung wurde gewählt, um die zeitweise Streuung der Daten aufgrund von Laständerungen im IB Netz zu verdeutlichen. Hierbei wurden 10 Messungen pro Dateigröße in zeitlichen Abständen von mehreren Minuten durchgeführt.
In beiden Fällen wurde, um eine maximale Performance zu erhalten, ein Striping von 8 für die Testdatei im Lustre-Dateisystem gewählt. Bei der Lösung mit IOFWD-Unterstützung wurde die Datei gleichmäßig auf 8 IOFWD Server, die sich auf unterschiedlichen IO-Knoten befanden, zusätzlich von IOFWD gestripet. Man sieht in den Abbildung zum Lesen der Datei zunächst einen Caching-Effekt für kleinere Dateigrößen für die Lösung ohne IOFWD. Sobald nicht mehr vom Lustre Client gecached wird, fällt die Performance hinter der IOFWD-Lösung zurück. Das Schreiben einer Datei mit IOFWD ist bei größeren Dateien schneller.

Abbildungen 5.6 auf Seite 70 und 5.7 auf Seite 70 zeigen den Transfer einer Datei in

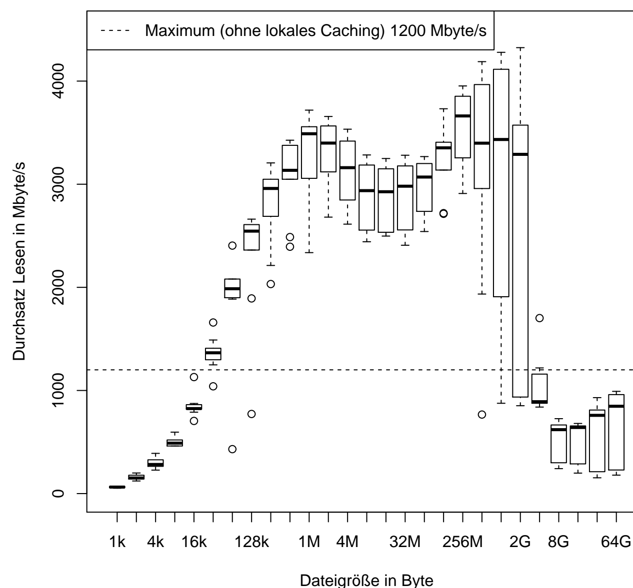

Abbildung 5.2: Lesen ohne IOFWD über natives Infiniband Linux-zu-Linux

einzelnen 1-kByte-Blöcken. Mit IOFWD wird ab einer gewissen Dateigröße eine bessere Performance erreicht. IOFWD sammelt die Lese- und Schreibanweisungen von der

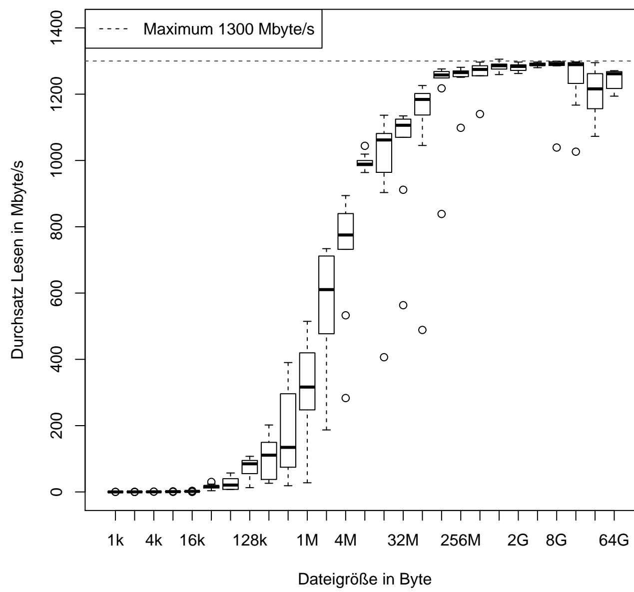

Abbildung 5.3: Lesen mit IOFWD über natives Infiniband Linux-zu-Linux

Applikation und schreibt sie dann in einem zum IOFWD Server. Dadurch wird eine höhere Bandbreite für den Datentransfer erreicht. Dies wird in den Abbildungen ab einer gewissen Dateigröße besonders für die Schreibvorgänge deutlich sichtbar.

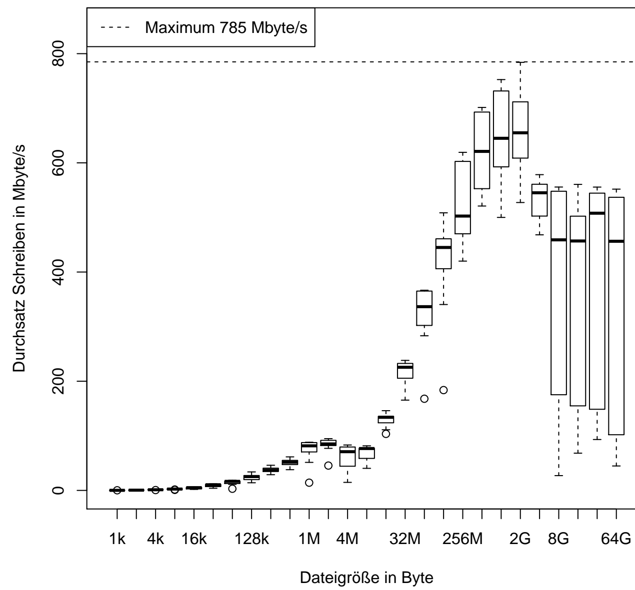

Abbildung 5.4: Schreiben ohne IOFWD über natives Infiniband Linux-zu-Linux

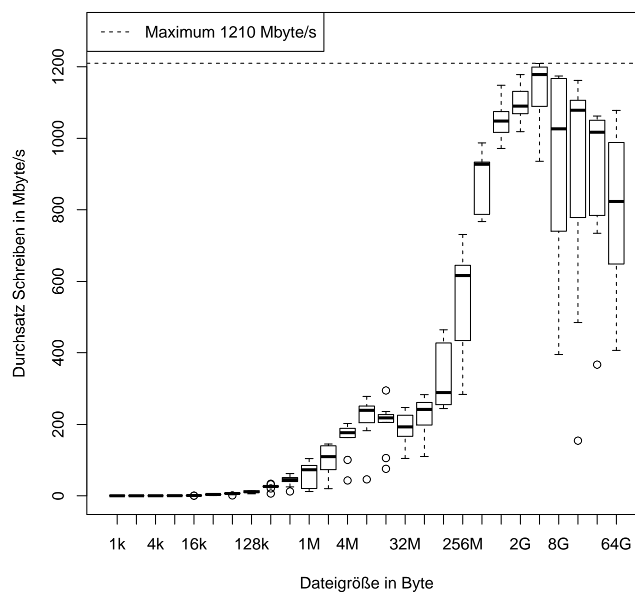

Abbildung 5.5: Schreiben mit IOFWD über Infiniband Linux-zu-Linux

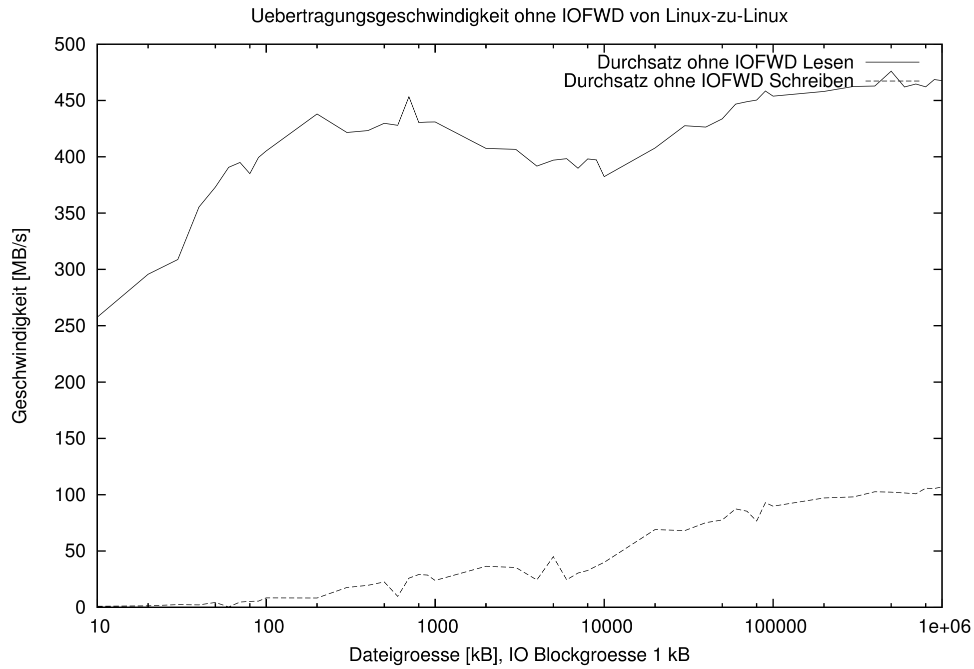

Abbildung 5.6: Lesen/Schreiben von einzelnen 1-kByte-Blöcken ohne IOFWD über Infiniband Linux-zu-Linux

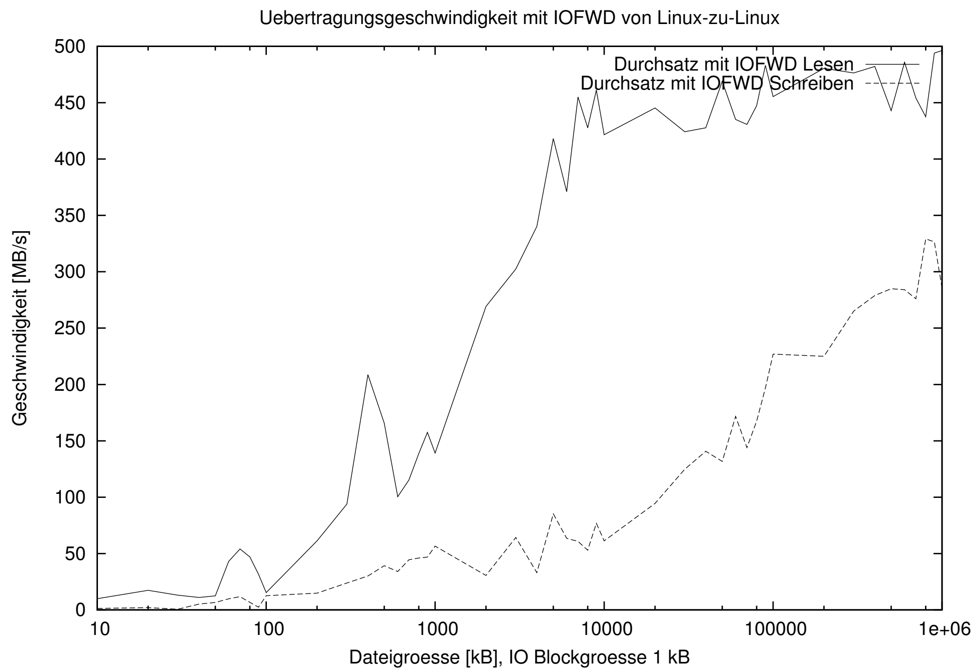

Abbildung 5.7: Lesen/Schreiben 1-kByte-Blöcke zusammengefasst mit IOFWD über Infiniband Linux-zu-Linux

6 PGAS-Run-time mit IOFWD

Der Grund, warum PGAS auch seinen Weg in diese Arbeit gefunden hat, ist die Verwendung des GASNet Frameworks in UPC. Auf diese Weise können UPC und IOFWD ein einziges GASNet Framework verwenden. Dies ist durchgeführt worden und dieses Kapitel beschreibt das Vorgehen dabei und zeigt auch Performance-Werte.

6.1 Was ist PGAS ?

Partitioned Global Address Space (PGAS) ist ein paralleles Programmiermodell [36]. Es läßt sich wie folgt in bestehende Programmiermodelle einordnen:

- Message Passing: z.B. MPI
- Shared Memory: z.B. OpenMP
- PGAS/Verteiltes Shared Memory: z.B. UPC.

Hier wird angenommen, daß ein globaler Speicheradressraum logisch aufgeteilt und jeweils ein Teil davon lokal zu einem Prozessor sind. Das Besondere an PGAS ist, daß Teile des Adressraums eine Affinität zu einem bestimmten Thread haben. Weiterhin ist es ein Vorteil von PGAS große Datenmengen in den verteilten Speichern vieler Prozesse unterzubringen, ohne daß dabei eine einheitliche Beschreibung der Daten verloren geht. Das PGAS-Modell ist auch die Grundlage für Unified Parallel C (UPC).
PGAS-Sprachen bieten die Möglichkeit komplexe geteilte Datenstrukturen zu erzeugen. Konstrukte wie multidimensionale Arrays, die die Sprachen bieten, vereinfachen die Programmierung. Jeder Thread kann direkt Daten lesen/verändern, die von einem anderen Thread erzeugt wurden. Daten können generell als lokal oder global definiert werden. Der Programmierer kontrolliert das Layout. Die PGAS-Sprachen haben viele Gemeinsamkeiten, wenn sie auch in Details differieren. Man kann die PGAS-Sprache als Erweiterung der Grundsprache sehen. Die gewöhnliche Definition einer Variablen entspricht der Definition für den privaten Adressraum in der PGAS-Sprache. Die Definition für den globalen Adressraum erfordert das „shared“ Schlüsselwort. Die Unterstützung für verteilte Datenstrukturen ermöglicht lokale und globale Pointer/Referenzen. Einfache Zuweisungen wie `x[i] = y[i]` sind z.B. auf memcpy-Operationen in UPC und Array-Operationen in Titanium oder CAF zurückzuführen. Für die Synchronisation gibt es verschiedene Mechanismen (siehe Abschnitt 6.2 auf der nächsten Seite). PGAS-Sprachen unterstützen einseitige Kommunikation. Bei Netzwerken wie Infiniband wird eine put/get-Nachricht direkt mit RDMA gehandhabt, d.h. der Prozessor ist in die Speicheroperation nicht involviert und wird somit entlastet.

6.2 PGAS Sprache Unified Parallel C

UPC [37, 38, 39] ist eine explizite parallele Erweiterung von ISO C. Es ist eine „partitioned shared memory“ parallele Programmiersprache. Das Ausführungsmodell von UPC erfolgt auf SPMD-Art und -Weise. Die Synchronisation erfolgt durch

- Barriers
- Locks
- Speicherkonsistenzkontrolle.

Das UPC Speichermodell basiert auf zwei Arten von Pointern:

- Pointer-to-Shared-Speicher
- Private Pointer.

Der Pointer-to-Shared kann alle Orte im zwischen den Knoten geteilten Speicher referenzieren, es besteht jedoch eine Daten-Thread-Affinität. Affinität bedeutet im physikalischen Sinne, daß der physikalische Speicher der CPU benutzt wird, auf der der Thread läuft. Der Private Pointer kann seinen Bereich des geteilten Speichers referenzieren oder einen Bereich in einem eigenen privaten Adressraum. Statische wie auch dynamische Speicheralloziierung werden gleichsam für geteilten wie auch privaten Speicher unterstützt.
Wenn Affinität definiert wird, unterscheidet UPC zwischen skalaren und Array Daten. Alle skalaren Daten haben Affinität zum Thread Nummer 0. Für Arrays erlaubt UPC 3 Typen von Affinität:

- zyklisch (pro Element): aufeinander folgende Elemente des Arrays haben Affinität mit aufeinander folgenden Threads
- Block-zyklisch: das Array ist in Benutzer definierte Blocks aufgeteilt und die Blocks werden zyklisch unter den Threads aufgeteilt
- Blocks: jeder Thread hat Affinität zu einem kontinuierlichen Teil des Arrays. Die Größe des kontinuierlichen Teils wird zur Laufzeit bestimmt und gleichmäßig auf die Threads verteilt.

UPC bietet Anweisungen, um die Affinität von Threads auf Datenteilen festzustellen. Zusammenfassend kann man sagen, dass eine Anzahl von Threads auf einem globalen Adressraum operieren, der logisch partitioniert unter den Threads ist. Jeder Thread besitzt eine Affinität zu einem Teil des globalen Adressraums. Jeder Thread besitzt auch einen privaten Adressraum. Das UPC-Speichermodell plaziert die Abfolge von Operationen auf eine pro Thread-Basis. Es existiert kein Mechanismus, die Abfolge von Operationen zwischen Threads festzulegen [40], was eine Schwäche des Speichermodells hinsichtlich sequentieller Konsistenz [41] ist. Das UPC-Speichermodell ist in dieser Hinsicht ähnlich zur Prozessor Konsistenz [42]. Hinsichtlich der Konsistenz gibt es zwei Arten von Operationen im Speichermodell:

- strict
- relaxed.

„Strict"-Operationen in einem einzelnen Thread sind linear geordnet. „Relaxed"-Operationen sind geordnet im Verhältnis zu Strict-Operationen, aber untereinander ungeordnet. Relaxed-Operationen sind ungünstiger bezüglich der Konsistenz, bieten jedoch dem Compiler mehr Möglichkeiten zur Optimierung des Code. Der Mangel einer linearen Ordnung von allen Operationen ist eine Schwäche im Vergleich zur Prozessor-Konsistenz. Die Einschränkung bei lokalem Lesen z.B. eines Threads der einen Wert liest, den er zuvor geschrieben hat, sind deutlich stärker als das Lesen eines Wertes, der von einem anderen Thread geschrieben wurde. Ein lokaler Read eines Wertes gibt immer den aktuellen Wert zurück. Ein entfernter Read (Wert wurde von einem anderen Thread geschrieben), muss der Ordnung der Operationen folgen, die der andere Thread durchläuft. Relaxed-Write-Operationen haben einen speziellen Status. Man kann den Schreibvorgang als eine Familie von Schreibvorgängen betrachten. Der Originalschreibvorgang auf einem Thread und eine virtuelle Kopie des Schreibvorgangs auf den anderen Threads, die diesen Vorgang beobachten. Es steht jedoch nicht fest in welcher Reihenfolge unterschiedliche Relaxed-Schreibvorgänge von den Threads betrachtet werden. Ein Relaxed-Read benötigt solche virtuellen Kopien auf den anderen Threads nicht, dieser ist ja nur für die Reihenfolge des eigenen Threads relevant. Die Problematik der Sichtbarkeit des Schreibvorgangs für andere Threads wird bewusst in Kauf genommen, um die Performance mit dieser Asynchronität zu erhöhen.
Notify und Wait (sogenannte „Split-Barrier-Operationen") bringen Threads zu einer Übereinkunft der entfernt sichtbaren Operationen für jeden Thread. Wenn ein Thread sein Wait beendet hat, sind seine Sicht der Operationen anderer Threads auf den Stand von bis zum Notify gebracht. Trotzdem ändert sich an der Reihenfolge der Operationen der Threads nichts. Die Operationen bleiben ungeordnet.

6.3 PGAS Sprache UPC mit IOFWD und GASNet

6.3.1 Einführung

GASNet wird in dem Runtime Framework von UPC verwendet. Ürsprünglich werden in dem UPC Framework nur Rechenprozesse gestartet. Abbildung 6.1 auf der nächsten Seite zeigt, wo die Prozesse von UPC und IOFWD gestartet werden und wie diese miteinander verbunden sind. Damit IOFWD hinzukommen kann, erfolgten folgende Änderungen:

- Die IOFWD Server Prozesse werden vom UPC Framework (genauer UPC GASNet) gestartet
- Die Veränderungen an dem IOFWD GASNet sind auch in das UPC GASNet gekommen und sind UPC neutral integriert

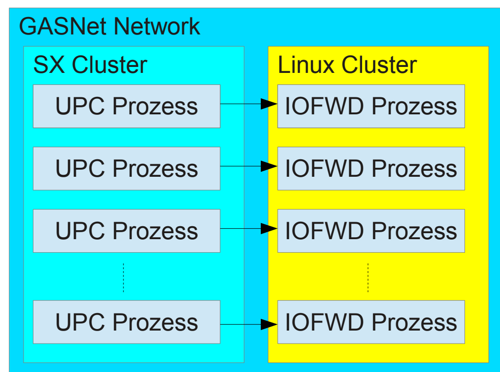

Abbildung 6.1: Rechen- und IO-Server Prozesse auf zwei Cluster verteilt

- UPC GASNet der SX Seite und IOFWD GASNet der Linux Seite müssen kompatibel sein
- GASNet UPC muss zwecks Starten von IOFWD Server Prozessen angepasst werden
- Die UPC Applikation erhält eine IOFWD Initialisierung
- Die UPC Applikation muss mit IOFWD kompiliert werden (siehe Abschnitt 6.3.2 auf der nächsten Seite).

Auf der SX-Seite ist alles statisch gelinkt, d.h. es entsteht eine Ausführbare, die mit IOFWD kompiliert wird und eine IOFWD-Initialisierung enthält. GASNet wird auch statisch hinzu gebunden. Auf der Linux-Seite ist alles dynamisch gebunden. Die IOFWD Server-Prozesse nutzen Shared-Bibliotheken wie die IOFWD und die GASNet-Bibliothek. Das Starten der Prozesse auf Linux und SX Cluster wird in Abschnitt 6.3.3 auf Seite 76 genauer ausgeführt. Die Veränderungen am UPC GASNet sind so geschehen, daß UPC, insbesondere die

- Speichertransferbereiche (Abbildung 6.2 auf der nächsten Seite),
- Barrier-Operationen (Abschnitt 6.3.5 auf Seite 80),
- die logische Nummerierung der Prozesse und die damit zusammenhängende Kommunikation (Abbildung 6.3.3 auf Seite 76)

nicht gestört werden. Bei den Speichertransferbereichen ist IOFWD an den Anfang des Transferbereiches von UPC geschoben worden. Der UPC-Transferbereich verschiebt sich nach oben. Das Verschieben ist eine gute Vorgehensweise, weil die Erzeugung der Transfersegmente davon unberührt bleiben kann. Die Adressen werden an geeigneter Stelle

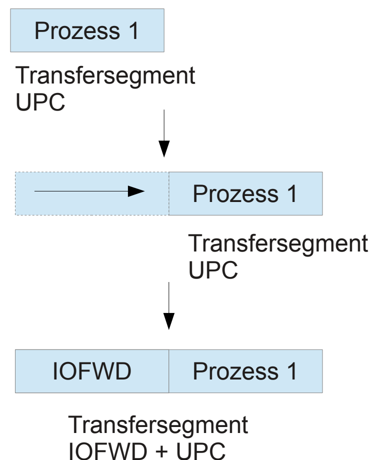

Abbildung 6.2: Einfügen des IOFWD Transfersegments

zum Verschieben aufgegriffen. Falls der UPC-Transferbereich in der Alloziierung verändert wird (insbesondere der Größe) hat dies keine Auswirkungen auf IOFWD oder UPC selbst.

6.3.2 Compile Wrapper

Es existieren Source-to-Source Compiler, die von der PGAS-Sprache in die native Sprache übersetzen, wie UPC in die Programmiersprache C. Dies hat den Vorteil, daß die PGAS Sprache von hoch optimierten, maschinenspezifischen Compilern übersetzt wird. Gewöhnlicher C Code kann für UPC ohne Änderungen übernommen werden, solange er nicht Schlüsselworte der UPC-Sprache enthält. UPC-Programme verwenden einen Compiler wie `upcc`, der dafür sorgt, daß z.B. zunächst der Source-to-Source Compiler aufgerufen wird und danach mit dem generierten Source Code der native Compiler. Der native Compiler des SX Knotens wurde in das UPC Runtime Framework einkonfiguriert. Dies musste über mehrere Ebenen der Konfiguration erfolgen. Man legt das zu verwendende Netzwerk wie z.B. UDP im Compiler-Aufruf mit der Option „-network udp“ fest. Dann wird beim Start der PGAS-Applikation das Run-Time Framework wie z.B. GASNet auf das entsprechende Netzwerk eingestellt. So kann exakt derselbe Programmcode auf verschiedenen Netzwerken verwendet werden. Der bereits vorgestellte Compile Wrapper wird nur um die Option des Prüfens auf das Netzwerk mit -network ergänzt. Nicht jeder Aufruf zum Source-to-Source Compiler ist gleich erfolgreich. Mit vielen Tests wurde festgestellt, daß hier in 1-2 von 10 Fällen der Aufruf nicht korrekt ausgeführt wird. Dies liegt hauptsächlich an gelegentlich vorkommenden Timeouts, da der Source-to-Source Compiler in der benutzten Installation vom SX-System aus auf einem Linux-System aufgerufen wird. Der Compile Wrapper muss hier auf Erfolg der

Kompilierkommandos mit upcc prüfen und das Kommando gegebenenfalls wiederholen.

6.3.3 Starten und Zuordnen von Prozessen

Abbildung 6.3 zeigt das parallele Starten von UPC-Rechenprozessen und IO-Server Prozessen. Beim Starten des UPC Runtime Frameworks wurde das Starten des IO-

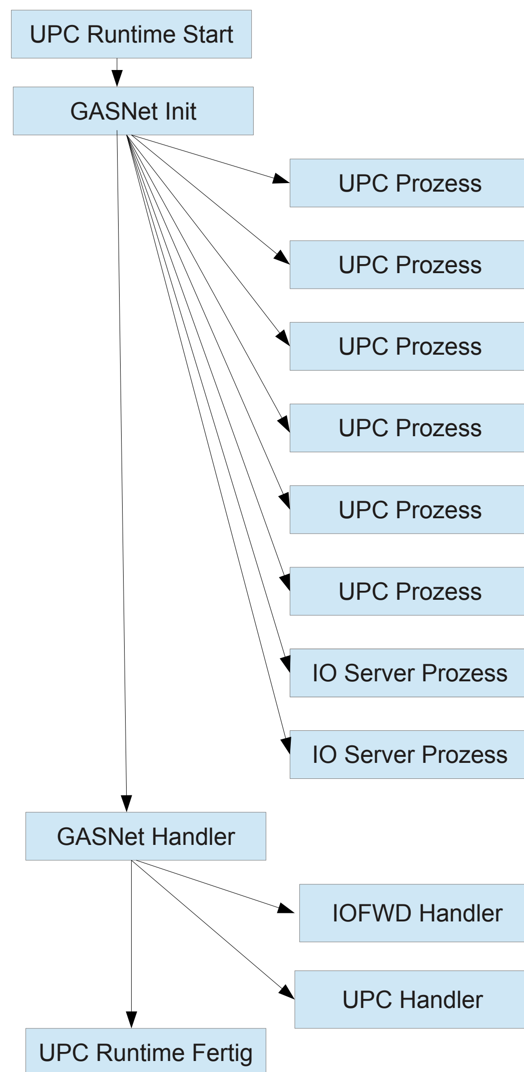

Abbildung 6.3: Starten der Prozesse in UPC

Framework IOFWD mit eingebaut. Ursprünglich wurde GASNet jeweils in UPC und IOFWD initialisiert. Dies wurde so geändert, daß die GASNet Initialisierung in UPC

genommen wird und IOFWD in UPC mit gestartet wird.
Es existieren zwei Gruppen von Knoten im GASNet Framework von UPC:

- Rechenprozesse
- IOFWD-Server-Prozesse.

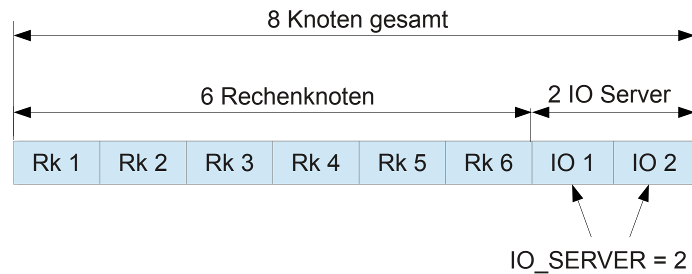

Abbildung 6.4: Liste der Orte für die Prozesse

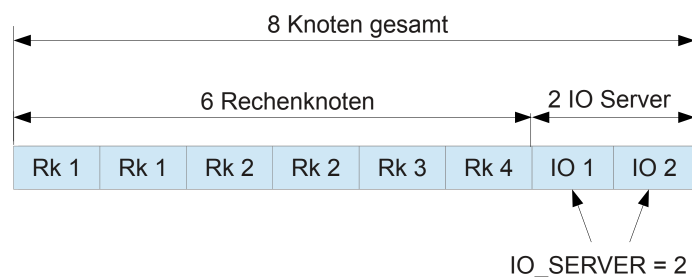

Abbildung 6.5: Prozessliste mit doppelten Nennungen in Reihenfolge

Alle Orte für die Rechenprozesse werden in einer Netzliste aufgeführt (siehe Abbildung 6.4). Hierin können auch Orte doppelt genannt werden (siehe Abbildung 6.5). Es sollte jedoch vermieden werden, wie in Abbildung 6.6 auf der nächsten Seite die Doppeltnennungen verteilt vorzunehmen, da eine gute Verteilung auf die IOFWD Server so nicht gewährleistet ist. Generell bezeichnet eine Variable des Systems die Anzahl der IOFWD Server. So wurde auch implementiert, daß GASNet diese zwei Netzlisten erstellt und

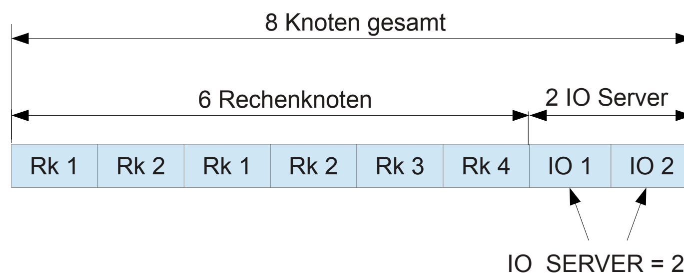

Abbildung 6.6: Prozessliste mit doppelten Nennungen nicht in Reihenfolge

die Prozesse entsprechend startet. Das Grundformat der Netzliste ist schon von GASNet vorgegeben. Das Hinzufügen von IO-Server-Adressen, wie auch die Erkennung und Auftrennung der Netzliste und das zielgerichtete Starten der Prozesse im Rechen- und IOFWD-Bereich wurden ergänzt.
Für diese beiden Knotentypen werden logisch zwei Netzwerke gebildet, denen einfache Netznummern von GASNet zugeteilt werden. Es wurde ein Algorithmus dafür implementiert, der diese Zuweisung durchführt.
Die Zuordnung der Prozesse geschieht wie in den Abbildungen 6.7 auf der nächsten Seite, 6.8 auf Seite 80 und 6.9 auf Seite 81. Hierbei werden die Rechenprozesse gleichmäßig über alle IOFWD Server-Prozesse auf allen IOFWD-Servern gleichmäßig verteilt. Eine Veränderung der Verteilung ist durch Manipulation der Netzliste für GASNet erreichbar. Gegenwärtig ist es so gehandhabt, daß die ersten Knoten in der Liste Rechenknoten und die letzten Knoten IOFWD Server-Knoten sind. Die Anzahl der IOFWD Server-Knoten gibt einen entsprechenden Hinweis für GASNet, wo die Trennung der beiden Gruppen erfolgt. Abbildung 6.10 auf Seite 82 zeigt, daß die Knotentypen unabhängig von einander gestartet werden und durch die logische Nummerierung eine Korrelation erhalten. Wichtig ist hierbei zu verstehen, daß vorher UPC mit Rechenprozessen alleinig ein Netz gebildet hat. Die IOFWD-Prozesse müssen in der Nummerierung so liegen, daß sie die UPC-Prozesse in ihrer Kommunikation nicht stören. Also wird die Nummerierung einfach am letzten Knoten fortgeführt und innerhalb von UPC sind die IOFWD Knoten nicht sichtbar. Die Verbindung von UPC und IOFWD-Prozessen muß nur von IOFWD aus sichtbar sein. Daher beinhalteten die Änderungen für GASNet UPC auch, daß für UPC nur die Rechenknoten sichtbar sind. Denn im Ausgangsfall wurden unter UPC alle verfügbaren Prozesse im GASNet-Netzwerk bearbeitet.
Bei der logischen Nummerierung bleibt die Original-Nummerierung der UPC-Prozesse erhalten. Die IOFWD Server-Prozesse werden aufsteigend angefügt und bleiben für UPC in ihrer Gesamtzahl unsichtbar. Da vorher die Operation auf die Gesamtzahl aller GASNet-Prozesse erfolgte, mußte dies auf die relative Anzahl der UPC-Prozesse ange-

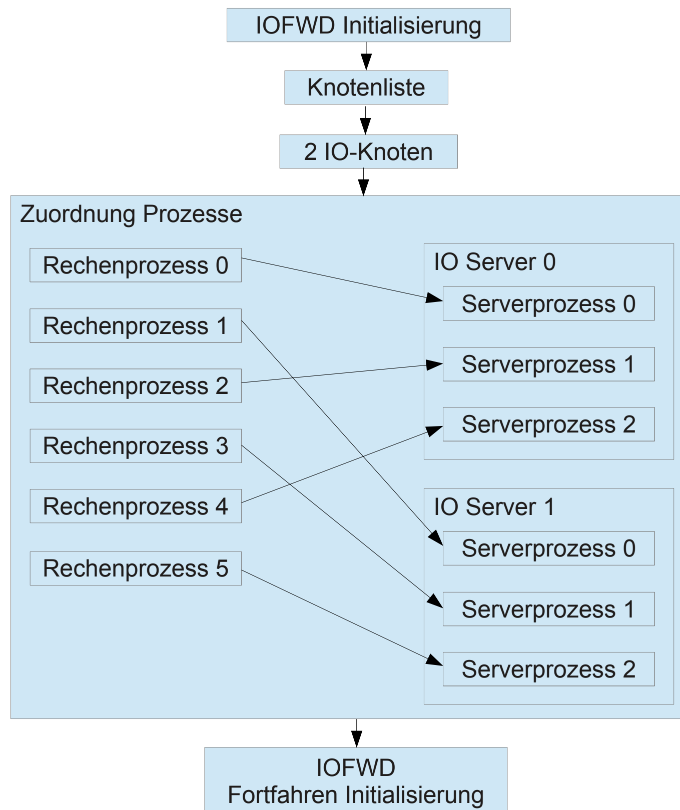

Abbildung 6.7: Starten der unterschiedlichen Prozesse: relative Prozessnummern

passt werden. Die Differenz der Knotennummer des UPC-Prozesses und der des IOFWD Server-Prozesses ist auch immer konstant gewählt. Als Beispiel sind die UPC-Prozesse mit der Gesamtzahl 6 von 0 bis 5 nummeriert und die IOFWD Server-Prozesse von 6 bis 11. Dabei korrespondiert Prozess Nummer 0 mit Prozess Nummer 6, Prozess Nummer 1 mit Prozess Nummer 7 ...

6.3.4 Handler

Handler sind Funktionen, die beim Senden von Nachrichten automatisch ausgelöst werden. UPC besitzt diese Handler und auch IOFWD hat Handler vorzuweisen. Das Hinzufügen von IOFWD-Handlern bedeutet lediglich eine gemeinsame Auflistung der Handler von UPC und IOFWD in UPC. Damit ist schon die Funktionalität durch GASNet gewährleistet.

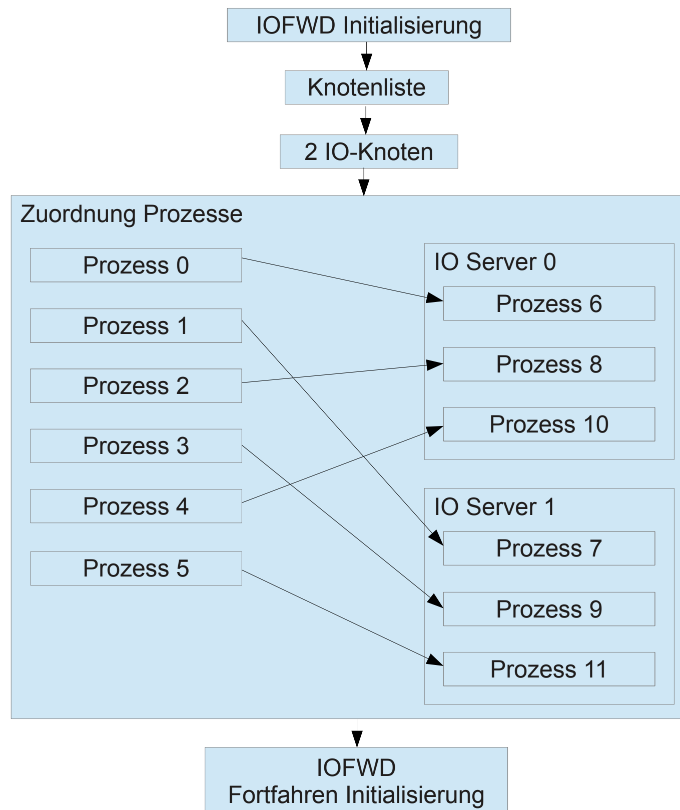

Abbildung 6.8: Starten der unterschiedlichen Prozesse: absolute Prozessnummern

6.3.5 Barriers

Barriers sind notwendig, um die Prozesse zu koordinieren. Dies spielt bei UPC insbesondere bei der Initialisierung eine Rolle. Jedoch existierten in der Ausgangssituation nur allgemeine für alle Prozesse gültige Barriers, welche die IOFWD und die UPC-Prozesse gleichsam bearbeitet hätten. Dies ist natürlich nicht gewünscht. Die Prozessgruppen für UPC und IOFWD sollen unabhängig Barriers verwenden können, sonst würde das Arbeiten der Prozessgruppen gestört. Hier wurden nun zwei Barriere-Typen erstellt, eine für IO-Knoten und eine für Rechenknoten. Die Startup-Sequenz von UPC wurde mit der IOFWD-Initialisierung integriert und die Barriers entsprechend konstruiert (siehe Abbildung 6.11 auf Seite 82). Das bedeutet, daß bei Nachrichtenaustausch (Haltepunkte, dann Weitermachen) nur entweder UPC-Rechenprozesse oder IOFWD Server-Prozesse betroffen sind. UPC-Rechenprozesse und IOFWD Server-Prozesse tauschen keine Nachrichten untereinander aus. Momentan tauschen auch die IOFWD Server keine Nachrichten untereinander aus. Jedoch ist es sinnvoll, schon jetzt eine Kommunikationsgruppe für die IOFWD Server einzurichten, da sie so zu einem späteren Zeitpunkt leicht weiter

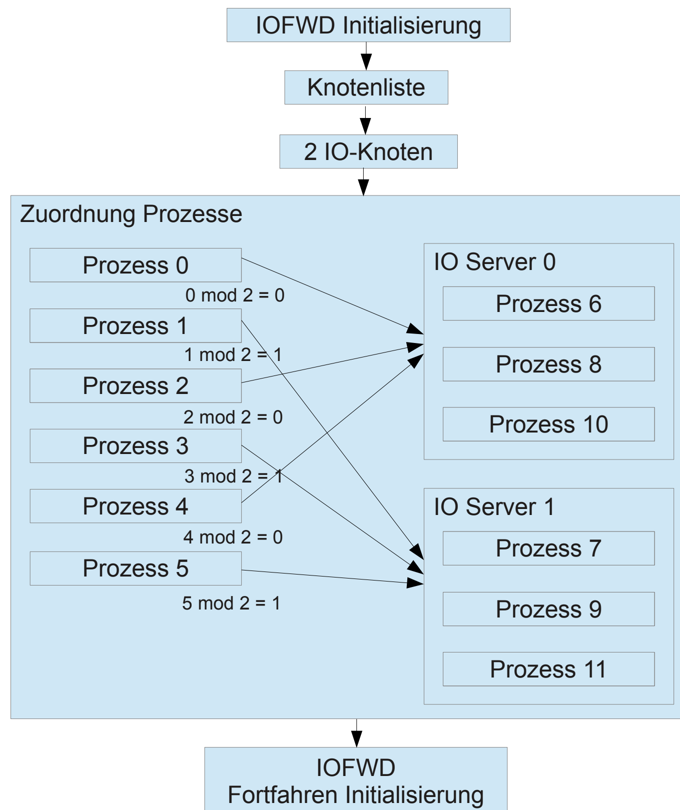

Abbildung 6.9: Starten der unterschiedlichen Prozesse: mod-Rechnung für Verteilung auf IO-Server

koordiniert werden können.

6.3.6 Benchmark

Um die Performance von UPC-Rechenknoten untereinander zu messen, wurde eine Benchmark verwendet, die das Kopieren innerhalb eines Shared UPC Arrays misst. Hierbei wird die Affinität des Speichers zu Threads genutzt (siehe Abschnitt 6.2 auf Seite 72) und die Array-Teile werden Block-zyklisch auf die Threads verteilt (siehe Abbildung 6.12 und 6.13). Abbildung 6.14 auf Seite 84 und 6.15 auf Seite 84 zeigen den Durchsatz für den Transfer von Array-Blöcken über GASNet UDP zwischen zwei UPC Threads. Hier ist schon bei 8 Mbyte Größe die maximale Transferrate erreicht.

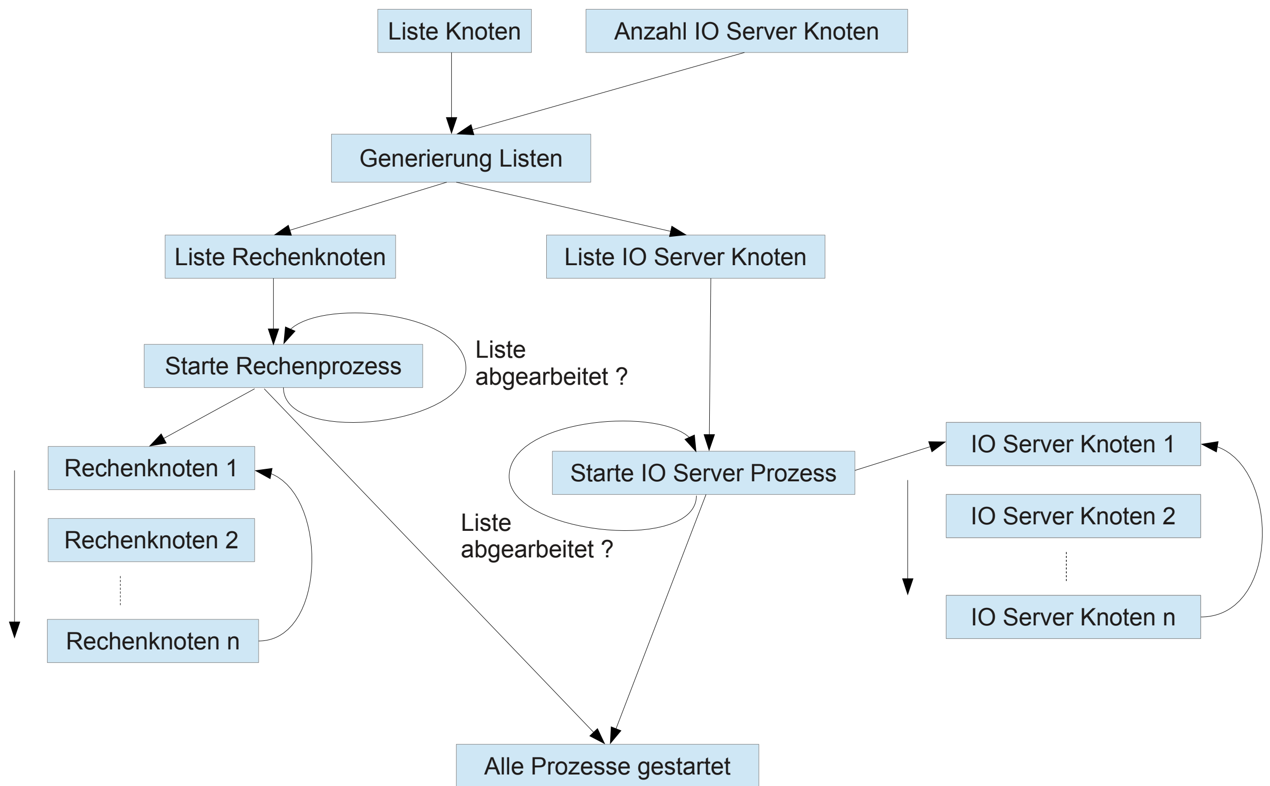

Abbildung 6.10: UPC-Initialisierung: Starten der unterschiedlichen Prozesse

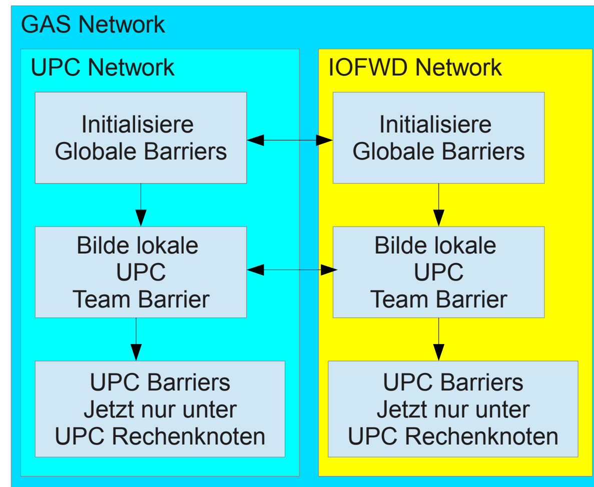

Abbildung 6.11: UPC-Initialisierung: Aufteilung der Barriers

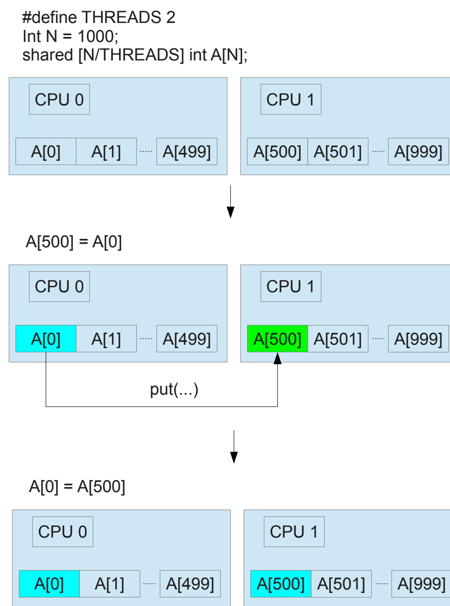

Abbildung 6.12: Shared Array-Zuweisung auf Knoten 0: put wird ausgelöst

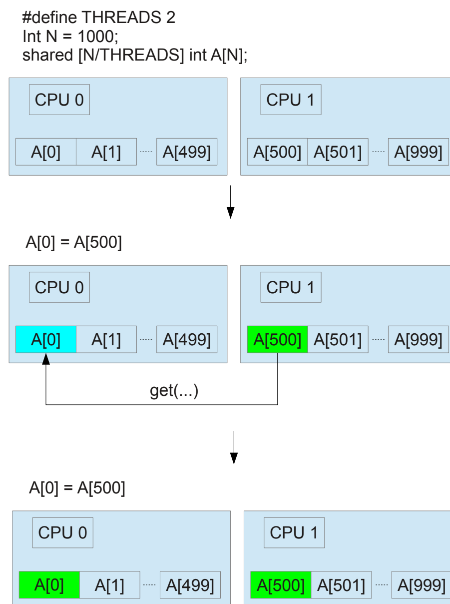

Abbildung 6.13: Shared Array-Zuweisung auf Knoten 0: get wird ausgelöst

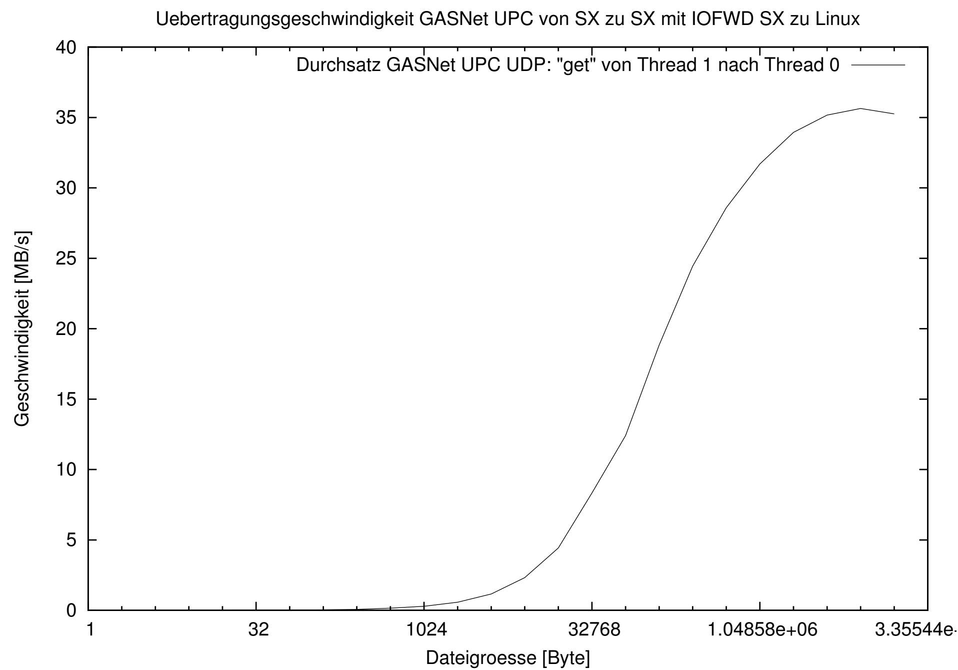

Abbildung 6.14: GASNet put-Operation von Thread 0 nach Thread 1
Schreiben von Thread 0 nach Thread 1

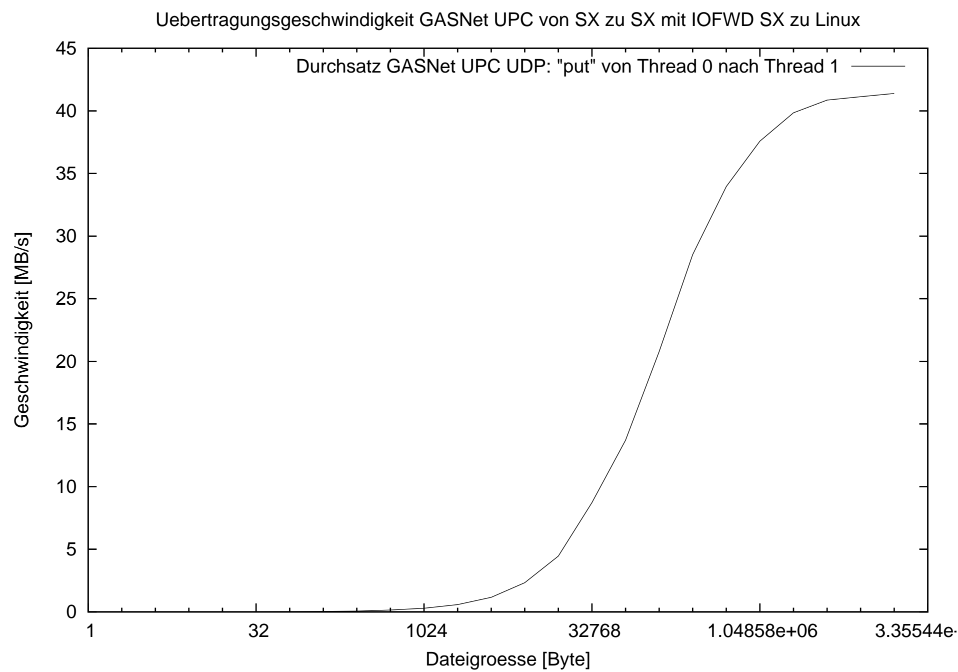

Abbildung 6.15: GASNet get-Operation von Thread 1 nach Thread 0
Schreiben von Thread 1 nach Thread 0

7 Simulations-Anwendung mit IOFWD

7.1 Beschreibung der Simulation

Diese Simulation [24] wird durch eine gekoppelte aero-akustische Applikation gezeigt (siehe Abbildung 7.1). Um die Generation von Schallwellen einerseits und deren Aus-

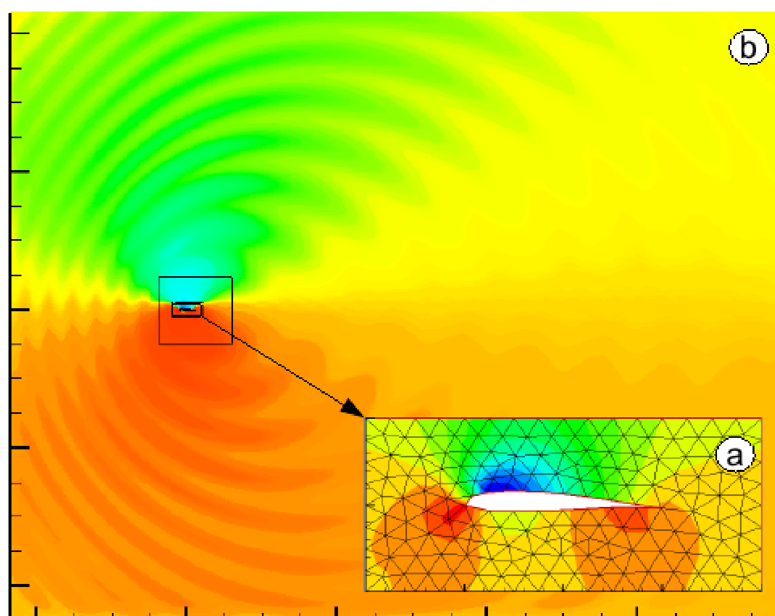

Abbildung 7.1: Allgemeine Konfiguration eine aero-akustischen Simulation. Um die Tragfläche befindet sich ein unstrukturiertes Gitter auf der der physikalische Fluß berechnet werden muß (a). Diese Domain ist eingebettet in einer Fernfeld Domain mit einem strukturierten Gitter, in der die Wellenausbreitung berechnet wird (b).

breitung andererseits effizient zu simulieren, ist ein Kopplungsschema notwendig, um unterschiedliche Berechnungen in dem jeweiligen Teil zu erlauben. Die direkte Simulation des multiskalen aero-akustischen Problems benötigt die Lösung des Strömungsfeldes einschließlich möglichst vieler physikalischer Strömungseffekte und die Lösung der Schallwellenausbreitung mit den Wellen als dominierendes Element des Interesses. So wird das Strömungsfeld mit den vollständigen Navier-Stokes-Gleichungen berechnet, wobei

es ausreichend ist, daß die Schallwellenausbreitung mit linearisierten Euler-Gleichungen gelöst wird. Im üblichen Setup gibt es nur ein kleines Objekt, das den Lärmschall in der Strömung generiert und die Schallwellenausbreitung wird in einem viel größeren Gebiet von Kilometern generiert. Es ist möglich, so eine räumliche Teilung beider Domains zu vollziehen, die unterschiedliche numerische Anforderungen hat. Nur ein relativ kleines Gebiet, das das Hindernis umgibt muß mit vollen Navier Stokes Gleichungen und einer hohen räumlichen Auflösung berechnet werden. Das große Fernfeld kann mit linearen Euler-Gleichungen mit einer wesentlich gröberen Diskretisierung berechnet werden. Die Berechnung der Tiefe des Details, notwendig für Strömungsfeldeffekte, durch die komplette aero-akustische Domain wäre eine Verschwendung von Rechenresourcen. So ist es besser, verschiedene Berechnungsarten durch die Kopplung von heterogenen Domains zu erlauben.
In dieser aero-akustischen Simulation wird heterogene Parallelität durch PACX-MPI benutzt. Die direkte Simulation von aero-akustischen Phänomen ist ein Multiskalen-Problem:

- Physikalische Effekte generieren Geräusche im Größenbereich von $m^{-2} - m^{-3}$
- Die Ausbreitung von Schallwellen geschieht im Größenbereich von $m^2 - m^3$.

Der Erzeugung von Schallwellen wird generiert von Navier-Stokes-Gleichungen. Für die Schallwellenausbreitung sind linearisierte Euler-Gleichungen ausreichend. KOP3D koppelt diese beiden Formen. Diese Form der Simulation wird durch Kopplung von unterschiedlichen Domains erreicht und besteht aus folgenden Teilen:

- Löser für strukturierte Gebiete (Fernbereich)
- Löser für unstrukturierte Gebiete (für komplexe Geometrien)
- Kopplung (unter Benutzung von Ghost Cells).

Es wird das ADER-Schema (Arbitrary high order using DERivatives) höherer Ordnung verwendet. Das ADER-Schema wurde entwickelt, um numerische Lösungen von Systemen mit hyperbolischen partiellen Differentialgleichungen zu berechnen bis zu einer (wenigstens theoretisch) beliebigen Ordnung. Weiterhin werden verschiedene Diskretisierungsmethoden verwendet:

- Discontinous Galerkin
- Finite Volume
- Finite Differences.

Die Discontinious Galerkin-Methoden sind eine Klasse von numerischen Methoden, um partielle Differentialgleichungen zu lösen. Sie kombinieren Eigenschaften von dem Finite Element und Finite Volume Framework. Die Finite Element-Methode ist eine numerische Technik, um Annäherungslösungen von partiellen Differentialgleichungen und Integralgleichungen zu finden. Die Finite Volume-Methode ist eine Form zur Repräsentation und

Evaluation von partiellen Differentialgleichungen in der Form von algebraischen Gleichungen. Die Finite Difference-Methoden sind numerische Methoden zur Annäherung von Lösungen für Differentialgleichungen unter Benutzung von Finite Difference-Gleichungen.
Das Kopplungs-Tool KOP3D verwendet das Koppeln in Raum und Zeit. Benachbarte Domains können deshalb bestehen aus:

- unterschiedlicher räumlicher Diskretisierung
- unterschiedlichen Zeitschritten
- unterschiedlichen Löser-Gleichungen.

Das Ziel dieser Simulation war es, heterogene Domains unterschiedlichen Architekturen zuzuordnen. Der strukturierte Programm-Code mit finiten Differenzen ist leicht zu vektorisieren und hat wegen der Regularität gute Performance auf Vektorarchitekturen. Der unstrukturierte Programm-Code mit Discontinous Galerkin macht hohen Gebrauch von Caching. Dieser läuft daher gut auf PC Clustern.
Bei den Berechnungen, die auf der SX-Plattform gemacht werden, läuft die KOP3D Software auf einem strukturierten dreidimensionalen Gitter. Dieses Gitter beinhaltet zellenbasierte Werte, die den Zustand des Gitters zum jeweiligen Zeitpunkt beschreiben. Die Simulation wird für 0.5 s simulierter Zeit durchgeführt.

7.2 Erstellen der Simulations-Applikation

Zur Neuerstellung der Applikation wurden der Compile-Wrapper für Fortran verwendet. Dabei wurde einfach der Aufruf für den Original Compiler durch den Compile Wrapper-Aufruf ersetzt. Die IOFWD und GASNet Build-Umgebung wurden schon auf der SX-Plattform vorinstalliert. Die Hauptroutine der Simulations-Applikation erhielt noch eine IOFWD Initialisierungs-Sequenz. Danach wurde die Simulations-Applikation ohne weitere Veränderung gebaut.

7.3 Durchführung der Simulation mit IOFWD

Ausgangspunkt der Simulation sind 2 Cluster, der SX-Cluster und der Linux-Cluster. Beide haben ein eigenes Dateisystem (siehe Abbildung 7.2 auf der nächsten Seite). Das SX-System schreibt ohne IOFWD in das lokale GFS Dateisystem. Die Performance sieht man in Abbildung 7.3 auf der nächsten Seite.
Wenn Daten von einem Cluster für den anderen Cluster benötigt werden, müssen diese Daten zwischen den Dateisystemen kopiert werden. Das kostet Zeit. Für das Lustre-Dateisystem existiert keine Unterstützung im SX-Cluster. IOFWD verändert die Situation nun so, daß der SX-Cluster Daten zum Linux-Cluster, dem Lustre-Dateisystem, schicken kann (Abbildung 7.4 auf Seite 89). Hier sollte etwas wichtiges angemerkt wer-

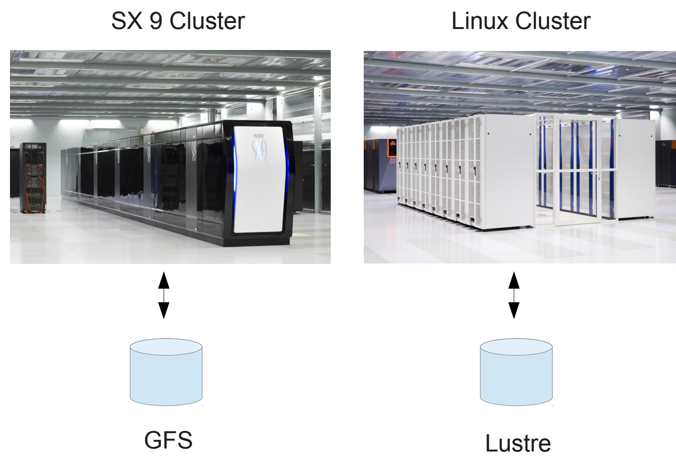

Abbildung 7.2: Simulation mit getrennten File Systemen

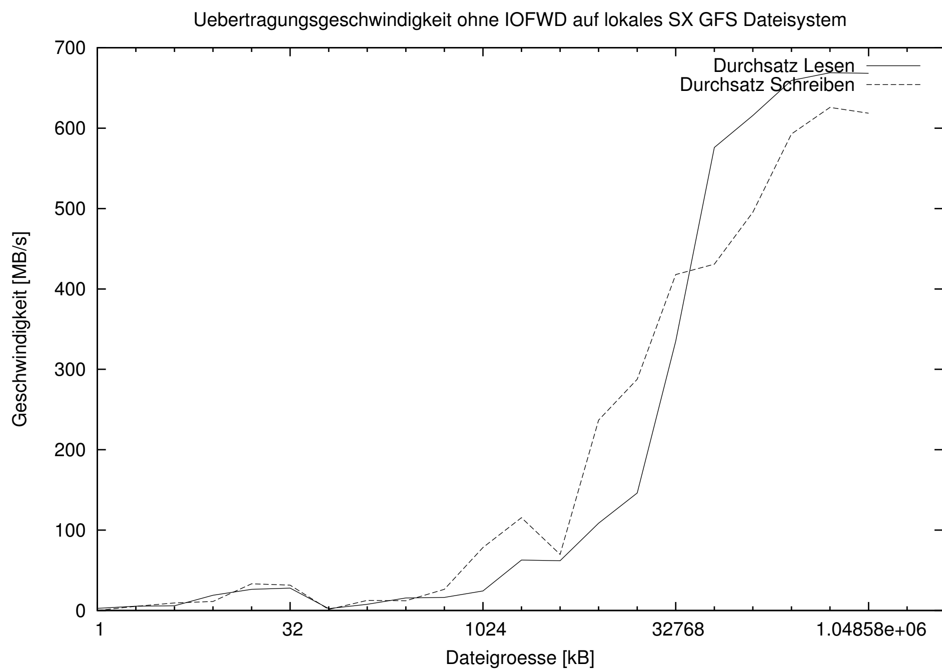

Abbildung 7.3: Datentransfer: lokales SX GFS Dateisystem

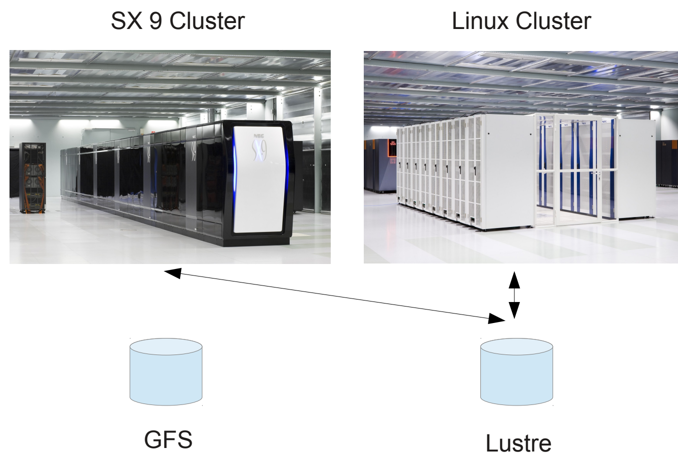

Abbildung 7.4: Simulation mit einem File System (Lustre)

den: Die Daten liegen in der falschen Endianness im Lustre-Dateisystem. Daher ist diese Art von Datentransfer nur für eine Wiederverwendung im SX-System gedacht, d.h. das SX-System kann diese Daten wieder lesen, denn sie behalten die korrekte Endianness für das SX-System. Die Daten sind nutzlos für das Linux-System. Dazu müßten die Daten Endianness-konvertiert werden. Das ist jedoch eine Payload-Konversion von Endianness-Daten, wozu IOFWD, zumindest momentan nicht gedacht und in der Lage ist. Payload-Konversionen erfordern eine genaue Kenntnis der Struktur und des Typs von Daten. Auch hier sei angemerkt, daß bestimmte Typen von Daten wie reelle Zahlen teilweise mit erhöhtem Aufwand konvertiert werden müssen, was wiederum die Performance bremst. Ideen und Konzepte hierzu befinden sich im Kapitel 9 auf Seite 100. Abbildung 7.5 auf der nächsten Seite zeigt den Aufbau der Simulation. Es wird zunächst mit 2 SX-Knoten angefangen und dann sukzessive um einen SX-Knoten erhöht, bis 6 SX-Knoten erreicht sind. Es wird exemplarisch für 2 SX-Knoten alle transferierten Dateien gezeigt (Abbildung 7.6 auf Seite 91). Die Abbildungen 7.9 auf Seite 94, 7.10 auf Seite 94, 7.11 auf Seite 95, 7.12 auf Seite 95, 7.13 auf Seite 96 und 7.14 auf Seite 96 zeigen die Performance bei den verschiedenen Knotenanzahlen.

Der Lustre Client (IOFWD Server) cacht die Schreibvorgänge zunächst und schreibt diese dann raus. Für die skalaren und vektoriellen Dateien war genug Speicher im Lustre Client vorhanden und so waren auch keine Einbußen hinsichtlich der Übertragung sichtbar. Bei den größeren Restart-Dateien reichte der Speicher des Lustre Client aber nicht aus, um alles zunächst zu cachen. Das Cachen ist natürlich so nicht erwünscht und kann entsprechend per Lustre-Dateisystem-Parameter im Lustre Client reduziert werden. Es besteht auch die Möglichkeit, dass der IOFWD Server seine Schreibvorgänge mit dem Flag `O_DIRECT` durchführt, das ein Cachen im Linux Client verhindert.

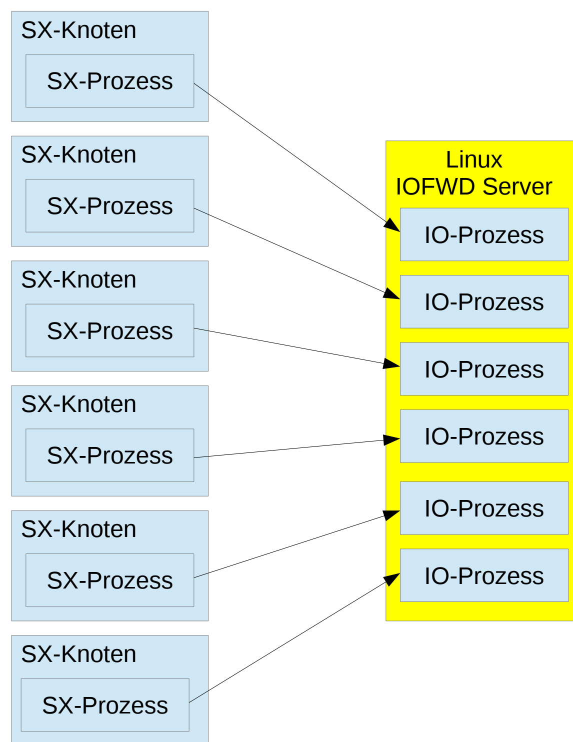

Abbildung 7.5: 6 SX-Knoten verbinden sich zu IOFWD Server

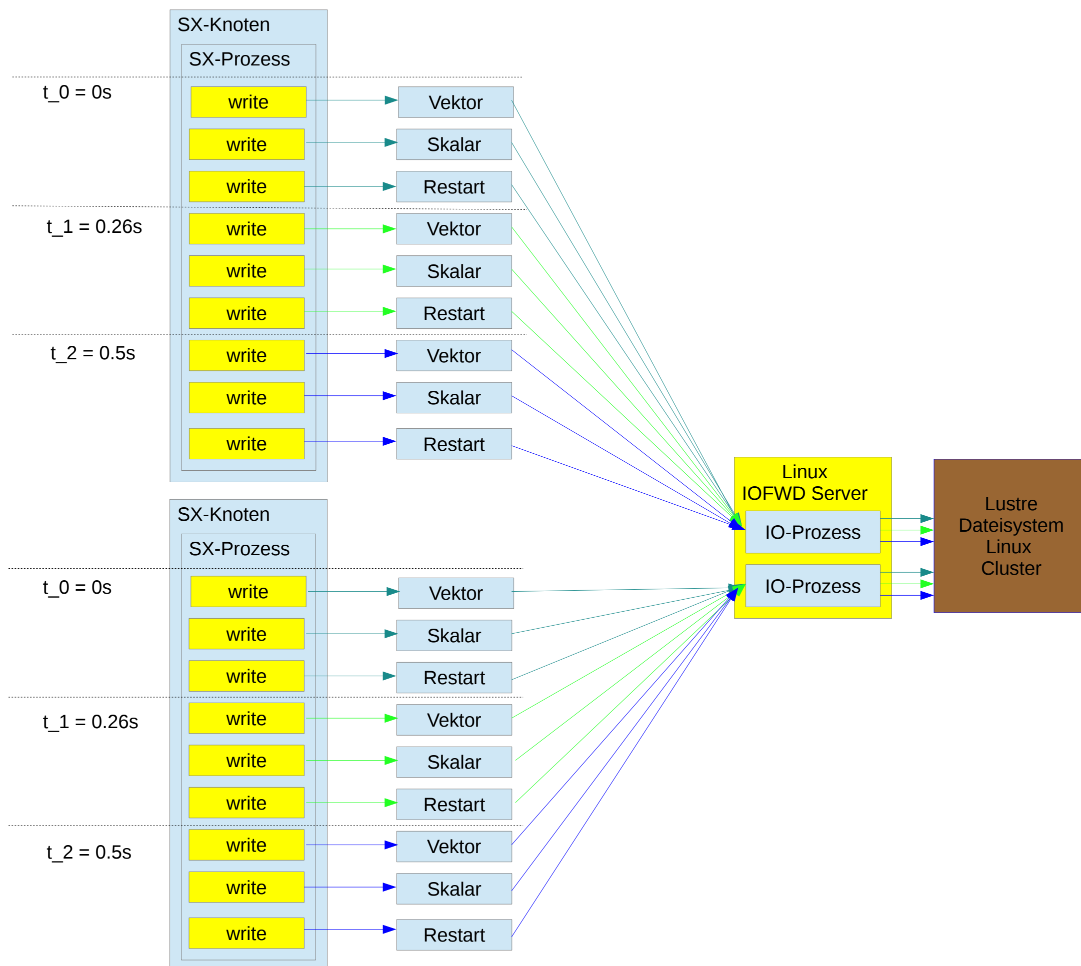

Abbildung 7.6: Zeitpunkte und transferierte Dateien für 2 SX-Knoten

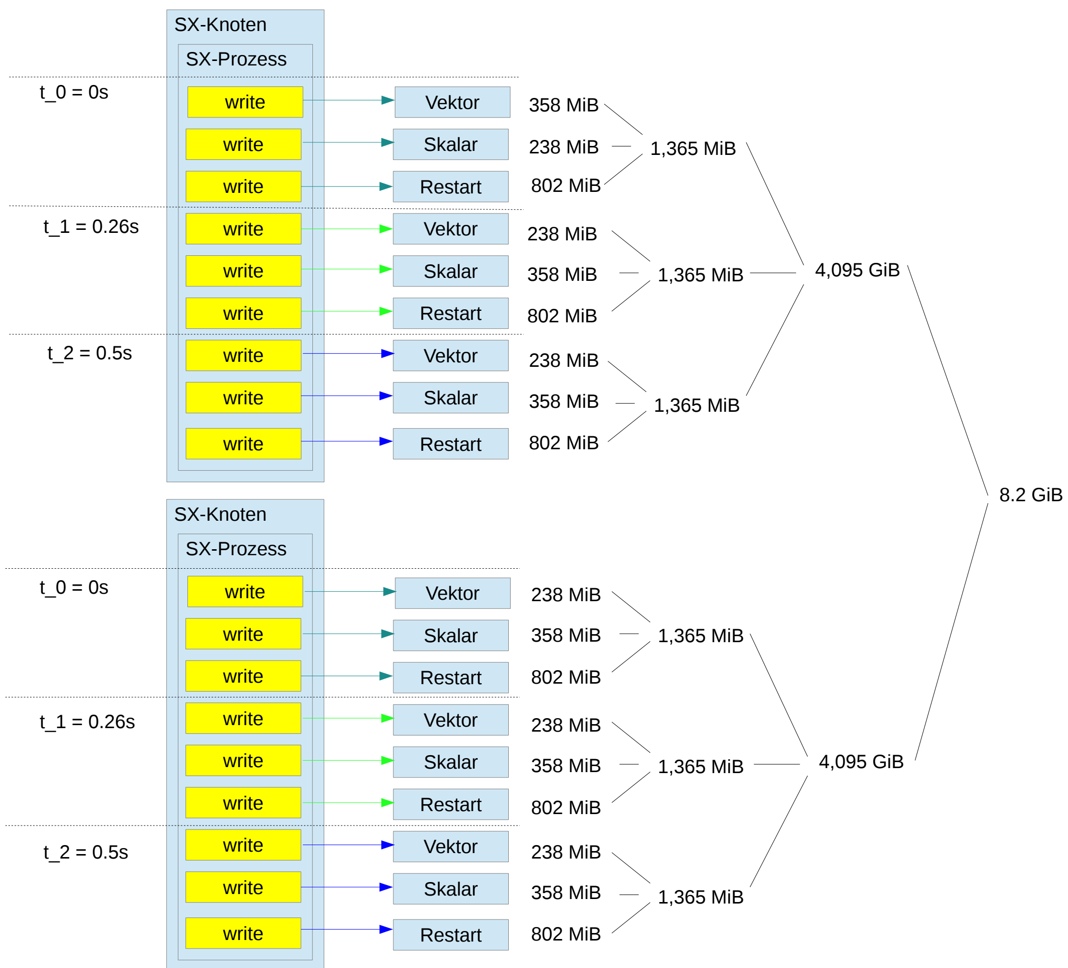

Abbildung 7.7: Transferierte Mengen für 2 SX-Knoten im Detail

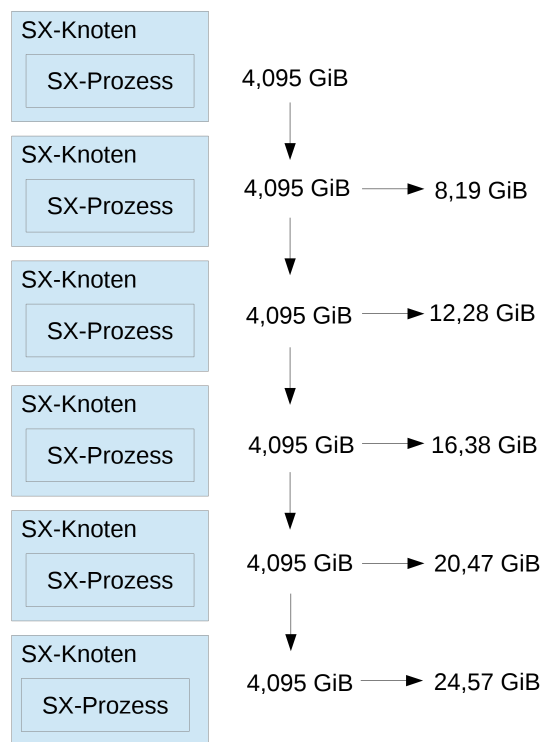

Abbildung 7.8: Transferierte Gesamt-Mengen für 2-6 SX-Knoten

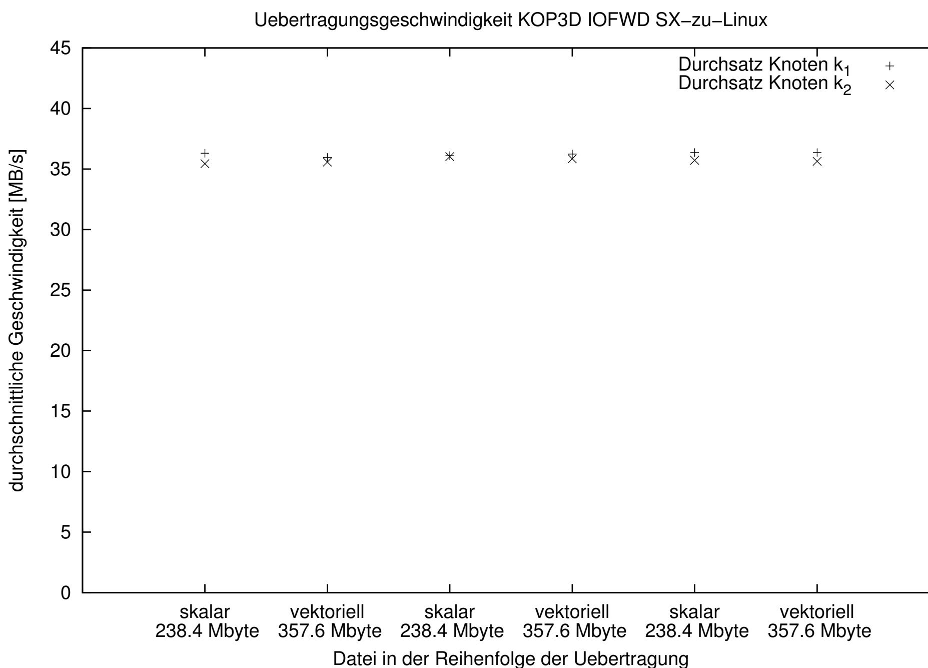

Abbildung 7.9: Datentransfer: Schreiben 2 SX-Knoten zu IOFWD Linux Server

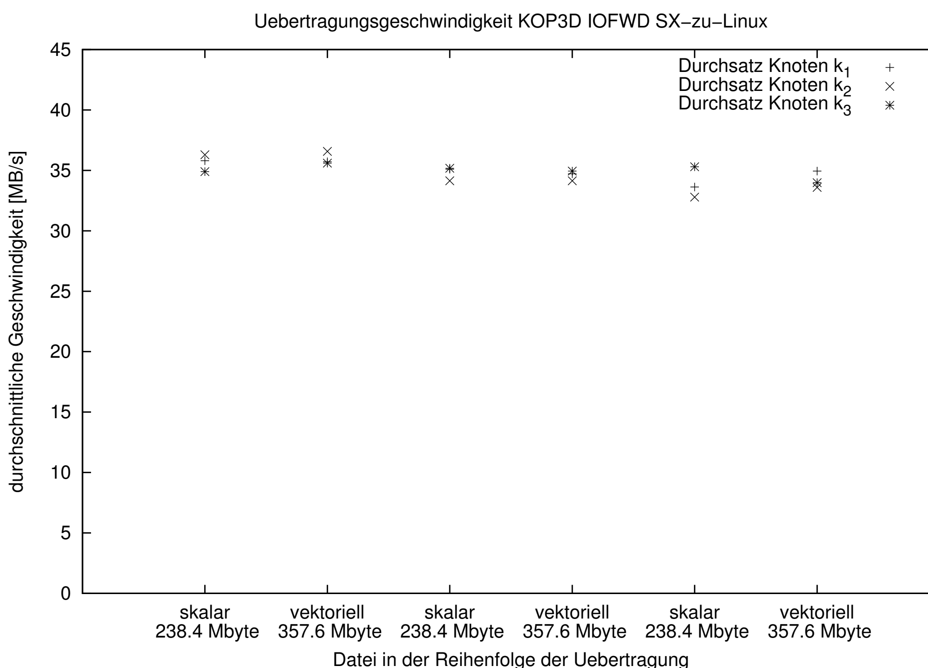

Abbildung 7.10: Datentransfer: Schreiben 3 SX-Knoten zu IOFWD Linux Server

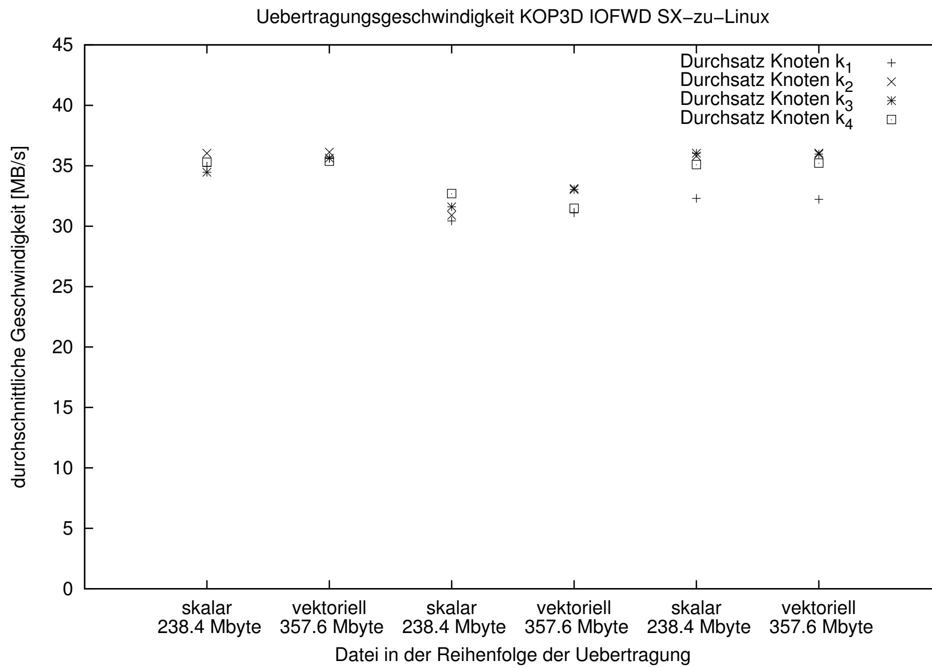

Abbildung 7.11: Datentransfer: Schreiben 4 SX-Knoten zu IOFWD Linux Server

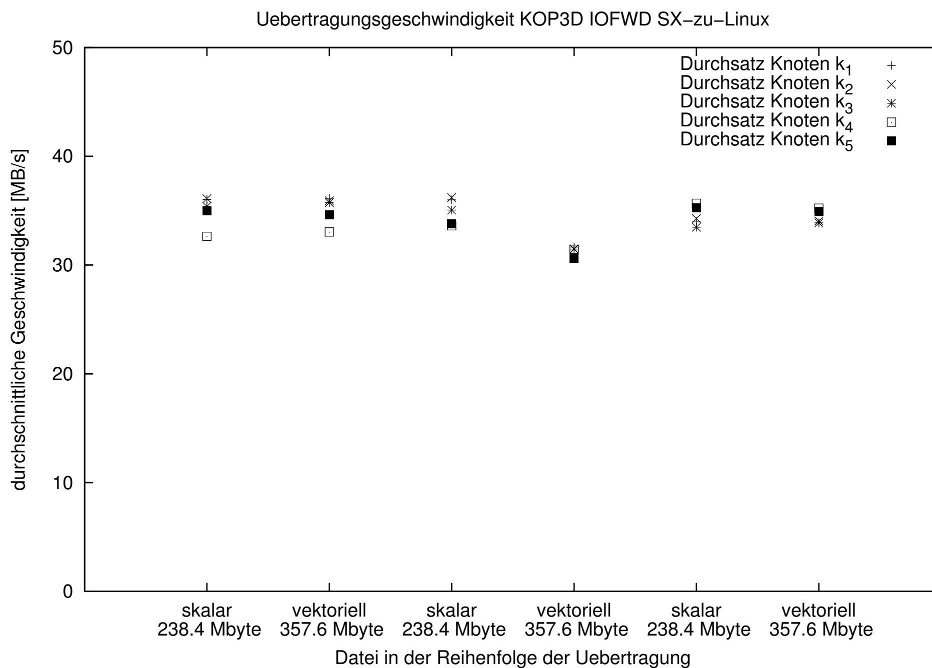

Abbildung 7.12: Datentransfer: Schreiben 5 SX-Knoten zu IOFWD Linux Server

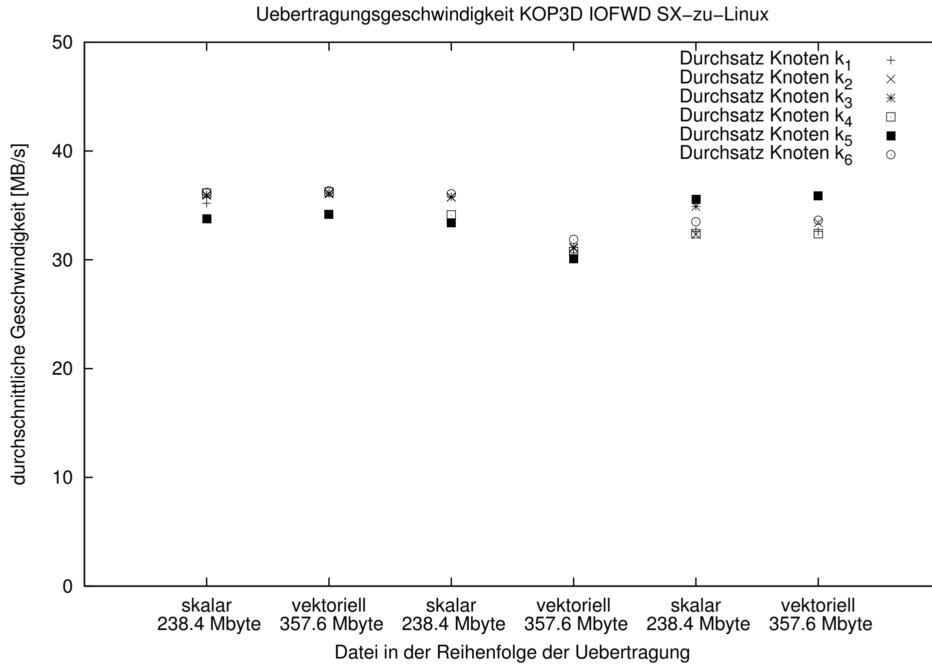

Abbildung 7.13: Datentransfer: Schreiben 6 SX-Knoten zu IOFWD Linux Server

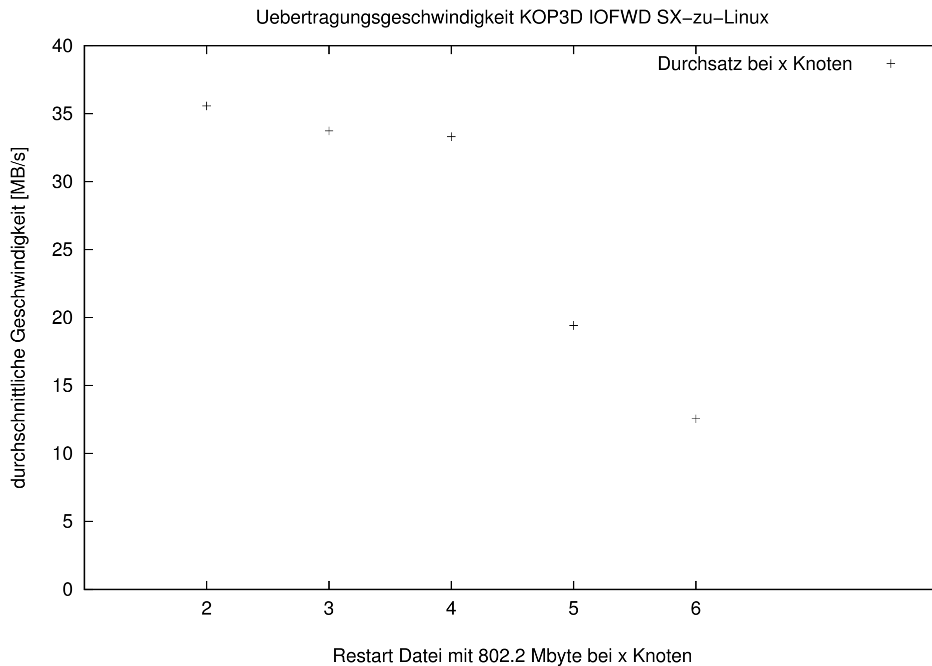

Abbildung 7.14: Datentransfer: Schreiben 2-6 SX-Knoten zu IOFWD Linux Server, gemittelter Restart-Durchschnitt pro Knoten

8 Schlussfolgerungen

IO-Forwarding in anderen Frameworks gab es schon während des Starts des Projekts IOFWD. Ein Projekt mit ähnlicher Zielsetzung wie IOFWD, IOFSL zeigt, wie bedeutsam software-basiertes IO-Forwarding in der HPC-Welt ist. Auch der möglichst universelle Aspekt der Lösung stößt auf großes Interesse. IOFWD ist ein Schritt in die richtige Richtung. Die vorgestellten Beispiele von Projekten mit unterschiedlichen IO-Charakteristika zeigen, wie unterschiedlich IO sein kann und wie dynamisch eine IO-Lösung sein muß. Einfluss auf das Konzept von IOFWD hat auch die Diskussion mit den Projektteilnehmern gehabt, wie man IO-Forwarding gewinnbringend in die Projekte einbringen kann oder anders gesagt wie kann man sich IO-Forwarding unter den genannten Projektbedingungen vorstellt und möglichst geschickt und einfach einbindet. Die Eingliederungsfähigkeit von IOFWD in ein System ist ein Maß für dessen Akzeptanz bei den Anwendern und auch für dessen Performance. Die Umgebung von IO-Forwarding, namentlich des Systems und der Hardware, haben mitbestimmt wo Modifikationen und Anpassungen im System zu machen waren. Die optimale Lösung wird als Einbindung einer IOFWD-System-Bibliothek gesehen, die sicher und performant im System arbeiten kann. Diese Lösung wurde angedeutet mit der Modifikation der System-Bibliotheken auf der SX-Platform, die die IOFWD Bibliothek benutzt haben. Besser ist es, der Maintainer der Distribution macht das Hinzufügen von IOFWD während des Erstellens der Bibliotheken. So ist eine stabile und ortsfeste Integration gewährleistet.

Die Verschiebung von IO-Verarbeitung auf Speichertransferleistung ist ein Paradigma von IO-Forwarding mit IOFWD. Das Übertragungsverhalten von IOFWD hängt auch mit dem begrenzten Arbeitsspeicher zusammen. Hier spielt die Auslastung des Speichers auf den Systemen (IO-Client und IO-Server) eine wesentliche Rolle bei der Übertragungsqualität von IO. Viel Speicher ist immer weniger ein Problem. Die Rechenknoten hinterlegen ihren IO einfach nur im Speicher und holen ihn auch dort wieder ab. Das Netzwerktransportsystem muss besser werden dafür. Eine baumartige Struktur zu den IO-Knoten hin ist mehr als notwendig.

Der Dialog mit Anwendern und Projektteilnehmern zum Thema „Wie bringe ich IOFWD am leichtesten mit einer bereits existierenden Anwendung zusammen?“ ergab den Abschnitt über die Injektion von Code in die Anwendung, die IOFWD automatisch mit startet. Injektion von Code wird ursprünglich zum Debuggen und Fixen von Anwendungen benutzt. Die Anwendung soll dabei möglichst durchlaufen und nicht gestört werden. Gleichzeitig soll die Anwendung nicht neu übersetzt werden müssen. Warum sollte dies auch nicht für IO-Forwarding gelten? Das Prinzip von IOFWD erlaubt diesen Ansatz. Das ist neu und sehr interessant auch für kompliziertere IO-Forwarding-Lösungen, die Pfade oder Bäume in ihrem IO-Weg durch schreiten und über Injektionen geleitet werden können.

Der Rechenknoten braucht eigentlich kein Dateisystem. Vielleicht ein Art lokales RAM-System gekoppelt mit einem geflashten Start-System. Die IO-Knoten haben die Dateisystem-Clients. Das ergibt deutlich weniger Clients und daher leichtere Verteilbarkeit des IOs innerhalb des IO-Systems auf das Dateisystem.
GASNet spielt eine wichtige Rolle in IOFWD. Ein wichtiger Teil der Funktionalität in IOFWD ist als eine Erweiterung des Global Address Space-Frameworks GASNet zu sehen. Es wurde an dem Netzwerk-Conduit und an dem Extended API von GASNet gearbeitet, um der Heterogenität die IO-Forwarding mit sich bringt, Rechnung zu tragen. Ein weiterer Aspekt an Heterogenität entstand auch durch den Verbund zweier unterschiedlicher Cluster, die sich in Hard- und Software wesentlich voneinander unterschieden. Man kann sagen, daß die minimale Anforderung zwei homogene Systeme, ein Rechensystem und ein IO-System, sind. Diese homogenen Systeme sind immer unterschiedlich aufgrund ihrer Aufgabenstellung.
Da das Paradigma der Verarbeitung von Daten in Richtung Speichertransfer aus Sicht des Rechenknoten geht, muß auch eine Art minimale Infrastruktur für den Transfer der IO-Daten geschaffen werden. Minimal waren hier ein fester Speicherbereich, der koordiniert unter allen Systemen konfiguriert wird. Einfachheit hat ihren Preis, die Identifikation der Daten ist natürlich nur von der Anwendung durchführbar. IOFWD transportiert Daten einfach nur. Auch hier hat sich gezeigt, dass Techniken wie Striping schon einen spürbaren Aufwand erfordern. Es gab viele Ideen, das Hauptaugenmerk lag jedoch auf der absoluten Transparenz für den Anwender. Ein gewisses Maß an Einfluß auf die IO-Operationen kann man ihm geben, jedoch ist dies im Rahmen der Arbeit eingeschränkt geblieben. Die Fehlermöglichkeit falscher Konfiguration von IOFWD steigt sonst zu überproportional.
UDP und Infiniband waren die hier betrachteten Netzwerkprotokolle. UDP konnte nicht überzeugen. Folgende Gründe kamen zum Tragen:

- umfangreicher GAS Framework UDP Conduit, der Remote Memory-Zugriffe letztendlich über einfaches Senden/Empfangen von Daten organisieren muss
- kleine maximale Transfergröße von UDP-Paketen
- angekommene Datenpakete werden nicht schnell genug weiterverarbeitet.

Infiniband hat sehr überzeugt. Dies liegt vor allem an der großen Übereinstimmung von Funktionalität des Speichertransferkonzeptes von GASNet und Infiniband. Das Extended API führt fast direkt Operationen auf Infiniband aus. Ohne große Umwege über Zwischenfunktionen. Die Messergebnisse sprechen für sich.
PGAS ist ein Paradebeispiel für die Benutzung eines Frameworks wie GASNet. Die PGAS-Sprache UPC, GASNet und IOFWD haben gut zusammengearbeitet. Hier konnte nur eine Beurteilung des funktionellen Zusammenspiels getroffen werden. UDP war leider nicht sehr performant. Gründe hierfür sind oben angeführt.
Die Anwendung der Simulation zeigte unter UDP wieder eine geringe Performance. Infiniband schnitt hier sehr gut ab. Die Simulation zeigte die leichte Anwendbarkeit von IOFWD. Es war leicht möglich, den vorgegebenen Source Code in Fortran mit dem

IOFWD-Compile-Wrapper zu übersetzen. Um IOFWD für die Simulation verfügbar zu machen, mußte auch ein spezielles Batch Script für das Batch-System für die MPI-Umgebung zur Verfügung stehen. Die Anwendung hat dann schließlich, auch nach Erweiterungen in GASNet, IO-Forwarding für die Simulation zur Verfügung gestellt.
Im wesentlichen sind die beiden Parallelisierungsstrategien MPI und PGAS betrachtet worden. Bei beiden konnte IOFWD punkten und IO-Forwarding bei Infiniband performant zur Verfügung stellen. Der Ansatz von IOFWD ist eher vorteilhaft für PGAS-Umgebungen, aber auch MPI-Umgebungen können leicht profitieren. Die funktionelle Aufteilung in Rechenfunktion und separater IO-Verarbeitung macht es leichter den Datenverkehr schneller zu optimieren und letztendlich auch die Performance besser planbar und einheitlicher über unterschiedliche Systeme hinweg zu steigern. Die Entwicklung von Testszenarien ist wertvoller, da sie für eine breitere Systemlandschaft anwendbar sind. Es wird auch ermöglicht einheitliche IO-Charakteristiken hervorzustellen, die nicht systemabhängig sind. Der Aspekt der Wiederverwendbarkeit tritt stärker hervor. Dies bedeutet schnellere Entwicklung und bessere Optimierung.

9 Ausblick

Gegenwärtig wird ein Projekt zur Erweiterung von IOFWD mit weiterer Funktionalität angegangen. Diese Funktionalität soll auch Gebiete außerhalb von IO beinhalten. Dabei werden IOFWD Server und IOFWD Client zu einem sogenannten Mediator zusammengeführt. Als Eingang nimmt der IOFWD Server die Daten/Befehle entgegen und übermittelt diese gegebenenfalls an den IOFWD Client-Teil, von dem die Daten/Befehle wieder ausgehen. Man kann sagen, daß die gegenwärtige Implementation von IOFWD Client und IOFWD Server Spezialfälle eines Mediators darstellen. Mediatoren könnten auch verkettet werden. Falls ein Mediator als Eingang mehrere andere Mediatoren hat, so muss auch das IOFWD Server-Konzept überarbeitet werden. Dies kann auch bedeuten eine Umordnung/Manipulation der Eingangsbefehle/-daten vorzunehmen, um diese an einen oder mehrere Mediatoren weiterzuleiten.
Die Erweiterung der Funktionalität könnte auch in nachfolgenden Gebieten liegen:

- Konvertierung von Formaten
- Direktion von Dateizugriffen
- Plausibilitätsprüfung der Zugriffe
- Sicherheit.

Die Konvertierung von Formaten erfolgt durch

- Konvertierung von Bildformaten
- Transformation von Matrizen
- Veränderung von Eingabe- und Ausgabeformaten
- Anpassung der Endianness.

Die Plausibilitätsprüfung der Zugriffe kann z.B. bedeuten, daß das Zielverzeichnis überhaupt verfügbar ist. Es kann im allgemeinen auch bedeuten, weitere Informationen zur Verfügung zu stellen, wenn ein Zugriff nicht möglich/gestattet ist:

- falscher Benutzer
- Zielsystem ist in Wartung und steht ab Zeitpunkt x wieder zu Verfügung
- Lizenz abgelaufen.

Ist der IOFWD Server multi-threaded und könnte somit auch mehrere IOFWD Clients gleichzeitig bedienen, würde dies für den UPC-Fall keine wesentlichen Modifikationen bedeuten. Jeder IOFWD Client muss dann am Anfang der Kommunikation einen Speicherbereich auf dem IOFWD Server erfragen, den er exklusiv beschreiben kann. Der IOFWD Server braucht für jeden Thread einen eigenen Speicherbereich in dem vom GAS Framework reservierten Speicher.
Für den Fall mit IOFWD Client/Server-Paaren müsste das GAS-Netzwerk weiter angepasst werden. Das Problem ist, daß der IOFWD Server bereits gestartet ist, wenn der IOFWD Client gestartet wird. Das GAS Framework GASNet initialisiert sich ja, indem die Prozesse auf allen beteiligten Knoten initialisiert und gestartet werden. Ein möglicher Weg wäre, die IOFWD Server zu einem GAS-Netzwerk zusammenzufassen und einen IOFWD Client Mitglied an diesem GAS-Netzwerk werden zu lassen. Hierbei ist zu berücksichtigen, daß z.B. im Falle von UDP jeder Knoten für alle anderen Knoten Buffer (zum Senden/Empfangen) anlegt. Bei einer großen Anzahl von Knoten kann dies beträchtlich sein. Hier könnten im IOFWD Client und IOFWD Server die lokalen Buffer bei Bedarf aus einem Pool von Buffern verwendet werden.
Das Hinzufügen von IOFWD Clients zum vorherig beschriebenen GAS-Netzwerk kann bei einer großen Anzahl von Prozessen einen beträchtlichen Verkehr erzeugen, da alle Knoten sich untereinander abgleichen müssten. Hier wären zentrale Stellen wie „Registrare“ (z.B. zum Austausch von administrativen Daten) sinnvoll, über die sich die IOFWD Clients in das GAS-Netzwerk einbinden. Die IOFWD Clients müssten unter anderem Informationen erhalten wie

- eigene logische Netzwerknummer
- Netzwerkadressen der IOFWD Server und deren logische Netznummer.

Nicht zwingend nötig wären die Informationen über die anderen IOFWD Clients. Der Registrar würde dem IOFWD Server die Informationen von den IOFWD Clients mitteilen. Der Vorteil logischer Netznummern zu verwenden liegt darin, daß die Programmierung des IOFWD Clients eine Unabhängigkeit vom verwendeten Netzwerk behält und mit der Zuordnung der logischen Netznummern zu Netzwerkadressen eine Präferenz für die IOFWD Server festgelegt werden kann. Dies könnte konkret bedeuten, daß die logische Netznummer 0 den ersten IOFWD Server bezeichnet, die Netznummer 1 den nächsten in der Reihenfolge u.s.w.. Die IOFWD Server könnten somit die ersten logischen Netznummern belegen. Diese Reihenfolge könnte unterschiedlich aussehen für unterschiedliche IOFWD Clients.
Auch denkbar wäre ein Hinzufügen eines intermediären Prozesses, der Anfragen von IOFWD Clients zunächst sammelt und dann auf die IOFWD Server verteilt. Hier könnte auch eine

- Umordnung
- Zusammenfassung

der IO-Anweisung erfolgen, die der Erhöhung der Performance dient. Dieser Zwischenprozess könnte auch über den Registrar ins GAS-Framework eingebunden werden. Hierbei würden dann die IOFWD Clients die Netzwerkadressen dieser Zwischenprozesse erhalten und die Zwischenprozesse die Netzwerkadressen der IOFWD Server.

Danksagung

Ich möchte mich hiermit auch bei Erich Focht (NEC), dem Projektleiter und Danny Sternkopf (ehemals NEC) für die gute Zusammenarbeit bedanken, die meine Arbeit ermöglichte. Ich möchte mich auch bei Herrn Küster, Herrn Bönisch vom HLRS und nicht zuletzt bei Herrn Professor Resch, dem Direktor des HLRS, und allen, die mich unterstützt haben, bedanken.

Literaturverzeichnis

[1] Erich Focht, Thomas Großmann, and Danny Sternkopf. I/O Forwarding on NEC SX-9. In *High Performance Computing on Vector Systems 2010*, pages 53–62. Springer-Verlag, 2010.

[2] NEC und HLRS. https://gforge.hlrs.de/projects/sxlinux/, 2011.

[3] Erich Focht, Jaka Močnik, Fredrik Unger, Danny Sternkopf, Marko Novak, and Thomas Großmann. The SX-Linux Project: A Progress Report. In *High Performance Computing on Vector Systems 2009*, pages 79–96. Springer-Verlag, 2009.

[4] Aris Rommel. PC Hardware Tuning: 30 Prozent mehr Leistung mit SSD-Festplatten, 2011.

[5] Philip Carns Parallel, Philip Carns, and Robert Ross. BMI: A Network Abstraction Layer for Parallel I/O. In *Proceedings of IPDPS'05, CAC workshop*, 2005.

[6] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, and Lee Ward. Scalable I/O Forwarding Framework for High-Performance Computing Systems, 2009.

[7] IEEE und Open Group für Unix. IEEE Std 1003.1-2008, 2008.

[8] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, 2009.

[9] Sandra Loosemore, Richard M. Stallman, and Andrew Oram. The GNU C Library Reference Manual, 2007.

[10] libsysio http://sourceforge.net/projects/libsysio/, 2009.

[11] Rajeev Thakur and Rajeev Thakur Ewing Lusk. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation, 2002.

[12] Kamil Iskra, John W. Romein, Kazutomo Yoshii, and Pete Beckman. ZOID: I/O-forwarding infrastructure for petascale architectures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 153–162, New York, NY, USA, 2008. ACM.

[13] Philip Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327. USENIX Association, 2000.

[14] M. Eisler. XDR: External Data Representation Standard - RFC 4506, 2006.

[15] FUSE : Filesystem in Userspace, Sourceforge.

[16] R. Shane Canon, Matt Andrews, William Baird, Greg Butler, Nicholas P. Cardo, and Rei Lee. GPFS on a Cray XT, 2006.

[17] Ron Brightwell, Trammell Hudson, Kevin Pedretti, Rolf Riesen, and Keith D. Underwood. Implementation and performance of Portals 3.3 on the Cray XT3. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, 2005.

[18] Ron Brightwell, Trammell Hudson, Kevin Pedretti, and Keith D. Underwood. An Accelerated Implementation of Portals on the Cray SeaStar.

[19] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 40:1–40:12, New York, NY, USA, 2009. ACM.

[20] Ralf Schneider. Identification of Anisotropic Elastic Material Properties by Direct Mechanical Simulations: Estimation of Process Chain Resource Requirements. In *High Performance Computing on Vector Systems 2010*, pages 149–159. Springer-Verlag, 2010.

[21] HLRS. http://segl.hlrs.de, 2011.

[22] Natalia Currle-Linde, Uwe Küster, Michael Resch, and Benedetto Risio. Science experimental grid laboratory (segl) dynamical parameter study in distributed systems.

[23] DEISA. http://www.deisa.eu, 2011.

[24] Harald Klimach, Sabine P. Roller, Jens Utzmann, and Claus-Dieter Munz. Parallel Coupling of Heterogeneous Domains with KOP3D using PACX-MPI, 2007.

[25] Thorsten Von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation, 1992.

[26] Shaun Clowes. Fixing/Making Holes in Binaries.

[27] James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. *SIGPLAN Not.*, 30(6):291–300, 1995.

[28] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 358–370, Washington, DC, USA, 2006. IEEE Computer Society.

[29] Chikeung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[30] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. PEBIL: Efficient static binary instrumentation for Linux. In *ISPASS*, pages 175–183, 2010.

[31] Paradyn Parallel Performance Tools. DynInstAPI Programmer's Guide, 2009.

[32] Paradyn Parallel Performance Tools. SymtabAPI Programmer's Guide, 2009.

[33] Paradyn Parallel Performance Tools. StackwalkerAPI Programmer's Guide, 2009.

[34] Paradyn Parallel Performance Tools. InstructionAPI Programmer's Guide, 2009.

[35] Paradyn Parallel Performance Tools. Dependence Graph API Programmer's Guide, 2009.

[36] Partinioned Global Address Space, 2010.

[37] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification, 1999.

[38] UPC Consortium. UPC Language Specification v1.2, 2005.

[39] The George Washington U, Phillip Merkey Michigan Technological U, Steve Seidel, Tarek El-ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. UPC: Distributed Shared Memory Programming, 2005.

[40] William Kuchera and Charles Wallace. Illustrative test cases for the UPC memory model. Technical report, 2003.

[41] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[42] J.R. Goodman. Cache consistency and sequential consistency. Technical report, University of Wisconsin–Madison, 1989.

[43] Øystein Thorsen and Charles Wallace. Automated Verification of UPC Memory Consistency, 2006.

[44] William Kuchera and Charles Wallace. The UPC Memory Model: Problems and Prospects. *Parallel and Distributed Processing Symposium, International*, 1:16a, 2004.

[45] William Kuchera and Charles Wallace. Toward a Programmer-Friendly Formal Specification of the UPC Memory Model. Technical report, 2003.

[46] Christopher Barton, Călin Cascaval, and José Nelson Amaral. A characterization of shared data access patterns in UPC programs. In *LCPC'06: Proceedings of the 19th international conference on Languages and compilers for parallel computing*, pages 111–125, Berlin, Heidelberg, 2007. Springer-Verlag.

[47] Christopher Barton, Călin Caşcaval, George Almasi, Rahul Garg, José Nelson Amaral, and Montse Farreras. Multidimensional blocking in upc. pages 47–62, 2008.

[48] John Reid. Coarrays in the next Fortran Standard, 2010.

[49] John Mellor-Crummey, Laksono Adhianto, William N. Scherer, III, and Guohua Jin. A new vision for coarray Fortran. In *PGAS '09: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, pages 1–9, New York, NY, USA, 2009. ACM.

[50] John Mellor-Crummey, Laksono Adhianto, and William N. Scherer III. A Critique of Co-array Features in Fortran 2008 Working Draft J3/07-007r3, 2008.

[51] Das Elektronik-Kompendium. UDP - User Datagram Protocol, 2009.

[52] Das Elektronik-Kompendium. TCP - Transport Control Protocol, 2009.

[53] Dan Bonachea and Dan Hettena. AMUDP Active Messages Over UDP, 2000.

[54] Odysseas Pentakalos. An Introduction to the InfiniBand Architecture, 2002.

[55] Hussain, Gupta, and Liu. Using OpenFabrics InfiniBand for HPC Clusters, 2006.

[56] Ashok Raj. InfiniBand™ Host Channel Adapter Verb Implementer's Guide, 2003.

[57] Vijay Velusamy and Changzheng Rao. Programming the Infiniband Network Architecture for High Performance Message Passing Systems. In *ISCA*. McGraw-Hill, 2003.

[58] Christian Bell and Dan Bonachea. A new dma registration strategy for pinning-based high performance networks. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 198.1, Washington, DC, USA, 2003. IEEE Computer Society.

A IO-Systemaufrufe

Systemaufruf	Anzahl Argumente	Return-Wert
access	2	Ja
chdir	1	Ja
chown	3	Ja
close	1	Ja
dup2	2	Ja
dup	1	Ja
fchdir	1	Ja
fchmod	2	Ja
fcntl	2-3	Ja
fchown	3	Ja
fstatfs	2	Ja
fstat	2	Ja
fsync	1	Ja
ftruncate	2	Ja
getcwd	2	Ja
getdents	3	Ja
link	2	Ja
lseek	3	Ja
llseek	3	Ja
lstat	2	Ja
mkdir	2	Ja
open	2-3	Ja
readlink	3	Ja
read	3	Ja
readv	3	Ja
rename	2	Ja
rmdir	1	Ja
statfs	2	Ja
stat	2	Ja
symlink	2	Ja
truncate	2	Ja
umask	1	Ja
unlink	1	Ja
utime	2	Ja

Systemaufruf	Anzahl Argumente	Return-Wert
write	3	Ja
writev	3	Ja

Tabelle A.1: Unterstützte IO-Systemaufrufe mit Anzahl Argumente

B PGAS Unified Parallel C

Das UPC-Speichermodell hat keine Kohärenz-Eigenschaft. Kohärenz besagt, daß es eine lineare Ordnung von Schreibvorgängen gibt, auf die sich alle Threads einigen. Diese Bedingung wird vom UPC-Speichermodell aus zwei Gründen nicht eingehalten:

- Schreibvorgänge von unterschiedlichen Threads sind nie geordnet, auch wenn es sich um eine gemeinsame Speicherstelle handelt.
- Relaxed-Schreibvorgänge in einem einzelnen Thread, auch wenn es sich um eine gemeinsame Speicherstelle handelt, sind nicht geordnet.

Ein Beispiel für 3 Threads t_0, t_1, t_2 wobei

- `x = 1` ein write von 1 auf x ist
- `x = y` ein read von y und ein write von y auf x ist

Alle writes sind strict und die anderen Operationen sind relaxed. x und y sind mit Null initialisiert:

t_0: `x = -1; x = 1;`

t_1: `if (x == 1) y = 2;`

t_2: `local_a = y; local_b = x;`

Thread t_1 beobachtet den write von Thread t_0 für x mit dem Inhalt 1. Daher wird y auf 2 gesetzt. Da Thread t_2 offensichtlich Thread t_1 write auf y beobachtet, kann man nicht daraus schließen, dass Thread t_2 auch den zweiten write auf x mit 1 beobachtet. x kann von Thread t_2 durchaus als -1 gelesen werden.
Ein weiteres Beispiel, dass die Problematik der Reihenfolge von Operationen innerhalb eines einzelnen Threads t_2 verdeutlicht:

t_0: `x = -1;`

t_1: `x = 1;`

t_2: `if (x == 1) y = 2 else z = 3;`

Thread t_2 führt write-Operationen auf unterschiedliche Variablen y oder z aus, abhängig davon welche x Variable gelesen wird. Die lineare Abfolge von Operationen innerhalb eines Threads t_2 wird also beeinflusst von Zugriffen auf eine Variable x, die von anderen

Threads wie t_1 und t_0 gesetzt wird. Das Problem ist, daß das UPC-Speichermodell auf Programmreihenfolge fußt, aber die Programmreihenfolge ist nicht wohldefiniert ohne ein zugehöriges Speichermodell.
In [43] wird ein Werkzeug vorgestellt, das die Konsistenz von UPC-Speicheroperationen prüft. Hierbei werden Informationen wie

- auslösender Thread
- Speichermodell
- Speicherort
- geschriebener/gelesener Wert

ausgewertet und es wird eine Ordnung der Operationen - falls möglich - auf Grundlage der UPC-Spezifikation festgelegt. Für eine entsprechende Lösung wird ein Graph ausgegeben. Der gerichtete Graph zeigt die Reihenfolge der Operationen für jeden Thread in Spalten an, wobei die Operationen mit gerichteten Kanten verbunden sind, die die Reihenfolge der Operationen anzeigen. Da der Graph eine temporäre Reihenfolge von Operationen anzeigt, ist die Existenz eines geschlossenen Umlaufs ein Anzeichen für eine Inkonsistenz (zyklischer Graph). Ein zyklischer Graph wird festgestellt, indem die Ausführungspfade auf UPC-Konsistenz geprüft werden. Ist der Graph UPC konsistent, so werden Kanten infolge von Transitivität entfernt.
Das UPC-Speichermodell mit seinen potentiellen Problemen wird auch in [44] angesprochen. Hier wird eine neue Definition des UPC-Speichermodells vorgeschlagen, die

- die Bedeutung der Strict UPC Konsistenz enger fasst
- die direkte Verbindung von betrachteten Verhalten eines anderen Threads und der Gültigkeit der Ausführung herstellt.

In [45] wird auch das UPC-Speichermodell analysiert. Es werden hierin 3 Axiome zur Verbesserung des Speichermodells vorgestellt (wobei $w(a,t)$ einen Schreibvorgang von Thread t an Adresse a, w generell einen beobachteten write darstellt):

- wenn Thread t_1 einen Lesezugriff an Adresse a durchführt, dann muss von t_1 ein `write` $w(a,t_2)$ von Thread t_2 beobachtet worden sein
- wenn t einen read auf write w durchführt, dann muss t w beobachtet haben
- wenn $w_1(a)$ und $w_2(a)$ beobachtete writes sind und $w_1 < w_2$ sind, dann kann t nicht länger w_1 lesen.

Die Zugriffsmuster auf Shared-Daten in UPC-Programmen werden in [46] zur Steigerung der Performance studiert. Hier wird unter anderem zur

- automatischen Erkennung und Privatisierung von lokalen Shared-Daten

- Schaffung weiterer Möglichkeiten für den Programmierer, Shared-Daten auf die Threads zu verteilen

- Verwendung von hybriden Architekturen

geraten. Bei hybriden Architekturen ist hier an ein Cluster aus SMP Systemen gedacht (wobei die Knoten mit Multi-Core-Einheiten bestückt sind). In diesem Fall kann eine Kombination aus Compiler und Runtime Support eine optimale Gruppierung der UPC Threads erreichen. Die Gruppierung soll dann so gestaltet sein, daß die Anzahl der lokalen Zugriffe erhöht und somit die Performance gesteigert wird.
Die Performance von UPC wird in [47] gesteigert, indem UPC erweitert wird mit der Möglichkeit, Shared Arrays in multiplen Dimensionen anzulegen. Auf diese Art und Weise soll eine bessere Kontrolle der Lokalität von Daten erreicht werden.

C PGAS Co-array Fortran

Co-array Fortran [48], auch bekannt als F– ist eine Erweiterung von Fortran 95/2003 für parallele Umgebungen. Der Fortran 2008 Standard beinhaltet nun auch coarrays. Ein coarray Fortran-Programm wird interpretiert, als wäre es um ein Vielfaches ausgeführt worden. Jede Kopie des Programms hat eine eigene Anzahl von Datenobjekten. Die Kopie wird auch „Image“ genannt. Die Images sind durchnummeriert und können untereinander angesprochen werden.
Die Variablen innerhalb eines Images können so definiert werden, daß sie über die anderen Images modifizierbar sind. Die Array Syntax von Fortran ist erweitert worden mit angehängten Unterbeschreibungen, die eine präzise Beschreibung zu Daten bietet, die über die Images verteilt sind. Es besteht auch die Möglichkeit zur Synchronisation zwischen Images, damit Variablen während einer definierten Phase eines Programms von anderen Images abgefragt werden können. Es können alle oder eine beliebige Untermenge von Images synchronisiert werden.
Um Daten, die von mehreren Images definiert oder referenziert werden, kontrolliert zugreifbar zu machen kann man auch „Locks“ verwenden. Wenn eine Passage von einem Image gesperrt wurde, kann diese nur von demselben Image wieder entsperrt werden. Wenn ein anderes Image nachfolgend auf ein Lock zugreifen möchte, während dieses gesperrt ist, wartet es, bis der Lock freigegeben wird.
Jedes Image hat eigene Eingabe- und Ausgabe-Einheiten. Wenn ein Image einen Integer-Ausdruck verwendet um eine Einheit zu adressieren, wird immer die Eingabe/Ausgabe-Einheit vom ausführenden Image benutzt. Die Verbindung von einer namentlichen Datei mit mehreren Images befindet sich außerhalb des Fortan 2008 Standard, aber es wird erwartet, daß dies in einem nachfolgenden technischen Report geregelt wird.
Eine Gruppe an der Rice Universität verfolgt einen alternativen Weg von coarray-Erweiterungen der Fortran-Sprache [49, 50]. Die Gruppe sieht verschiedene Unzulänglichkeiten im Fortran 2008 Standard bezüglich der coarray-Erweiterung wie:

- coarrays müssen über alle Images alloziert werden
- coarrays müssen als globale Variablen deklariert werden
- der coarray-Erweiterung fehlt jeglicher Begriff von globalen Pointern, die wichtig sind zur Bearbeitung von verketteten Datenstrukturen
- es gibt keine kollektive Kommunikation
- es gibt keine Mechanismen, die Latenz zu verstecken bei Veränderung von Daten auf anderen Images.

Daher wurde ein eigenes Design für Co-array Fortran geschaffen: Co-array Fortran 2.0.

D Unterschiedliche Art von Übertragung in GAS-Netzwerken

D.1 UDP

D.1.1 Aufbau

User Datagram Protocol (UDP) ist ein verbindungsloses Transport-Protokoll [51]. Das bedeutet, im Gegensatz zum verbindungsorientierten Protokoll entfällt der Verbindungsaufbau und Verbindungsabbau. Es findet direkt die Übermittlung der Daten statt. Es befindet in der Transportschicht des OSI-Schichtenmodells und hat eine vergleichbare Aufgabe wie das verbindungsorientierte Transport Control Protocol (TCP) [52]. UDP arbeitet verbindungslos und unsicher. Es ist Aufgabe der Applikation für eine sichere geordnete Übertragung zu sorgen. Der Vorteil von UDP ist, daß es einen viel kleineren Paket-Header als TCP beinhaltet. Weitere Eigenschaften von UDP sind:

- keine Fehlerbehandlung
- keine Zeitüberwachung
- keine Flusskontrolle.

Unter Fehlerbehandlung versteht man das automatische Wiederverschicken von verlorengegangenen oder korrupten Daten. Die Zeitüberwachung hat ein Zeitfenster, in dem Pakete angekommen sein sollten. Bekommt der Sender keine Bestätigung vom Empfänger innerhalb eines Zeitraums wird das Paket erneut gesendet. Keine Flusskontrolle heisst, dass die Datenrate zwischen Sender und Empfänger nicht gesteuert wird (z.B. ein schneller Sender überfordert einen langsamen Empfänger).
Die Implementierung des UDP-Netzwerks in GASNet trägt den Namen AMUDP [53]. „AM“ steht für Active Messages. Der Grund, warum UDP TCP vorgezogen wurde, ist die Verbindungsorientiertheit von TCP. AM erlaubt es Nachrichten zwischen beliebigen Endpunkten innerhalb einer Gruppe von Kommunikationssystemen auszutauschen. Würde man dieses Kommunikationsschema auf TCP anwenden, würde eine quadratische Anzahl von Verbindungen im Verhältnis zu der Anzahl der Prozesse benötigt werden, da TCP ein verbindungsorientiertes Protokoll ist, bei dem die Verbindungen während der Initialisierung von GASNet aufgebaut werden. Weiterhin benutzt TCP im Vergleich zu UDP beträchtliche Resourcen. Die TCP-Lösung würde daher nicht gut auf großen verteilten Systemen skalieren. UDP passt daher perfekt auf das AM-Modell.
AMUDP verwendet einen Startmechanismus für Applikationen, die dem Single-Program

Multiple-Data (SPMD) Prinzip folgen. Eine spezielle Start-Routine innerhalb von AM-UDP sorgt dafür, daß nach Angabe der Prozesse in einem Job und des Kommandoaufrufs die Applikation auf den Knoten entsprechend gestartet wird. Der Startmechanismus für Applikationen wurde dahingehend angepasst, dass zwei unterschiedliche Arten von Applikationen auf Knoten gestartet werden:

- Rechenapplikation
- IOFWD Server Applikation.

Die Konnektivität der Knoten untereinander wird erreicht, indem Übersetzungstabellen und Tags entsprechend eingerichtet werden. Weiterhin werden Services zur Verfügung gestellt, die Eingabe und Ausgabe zwischen Benutzerkonsole und Worker-Prozess herstellen können. Es gibt generell zwei Arten von Knoten:

- Master-Knoten
- Worker-Knoten.

In Abbildung D.1 sieht man die Kommunikation der Master- und Worker-Knoten untereinander. Die Knoten erkennen beim Starten, ob diese Master- oder Worker-Knoten sind.

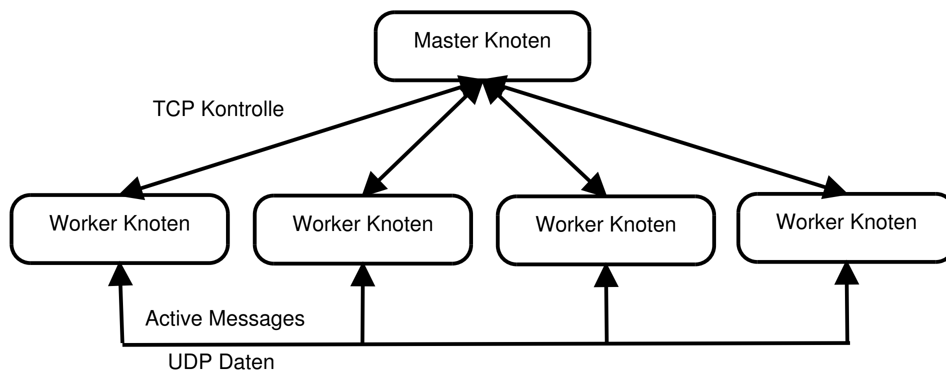

Abbildung D.1: Kommunikation der Master- und Worker-Knoten

Beim Startvorgang des Master-Knoten wird ein TCP Port gebunden, der zur Kommunikation zwischen Master- und Worker-Knoten verwendet wird. Die Worker-Knoten binden auch einen TCP Port über den dann mit dem Master-Knoten Initialisierungsdaten ausgetauscht werden. Der Master-Knoten sammelt alle Informationen von den Worker-Knoten, bündelt die Informationen und verteilt sie wieder auf die Worker-Knoten. Danach wird die Applikation auf den Worker-Knoten gestartet. Der Master-Knoten überwacht dann im folgenden die Worker-Knoten. Im Falle von MPI ist der Master-Knoten auch ein Worker-Knoten. Hier wurde der Startmechanismus so angepasst, daß die auf dem Master gestartete Applikation immer über ein `fork` Kommando wieder gestartet wird. Da IOFWD, mit Ausnahme von UPC, immer nur ein Netzwerk von zwei Knoten

verwendet (Applikation + IOFWD Server-Applikation), ist es möglich, daß die MPI-Umgebung der MPI-Applikation durch den fork erhalten bleiben. Die MPI-Umgebung ging bei der üblichen Methode mittels ssh verloren. Das fork-Kommando ist nicht nötig in UPC. Hier kann die Applikation auf dem Master für den Worker-Prozess komplett neu gestartet werden.

D.1.2 Übertragung

Das Active Message UDP Request/Reply-Netzwerk ist ein all-to-all-Netzwerk, d.h. jeder Knoten kann zu jeder Zeit mit jedem anderen Knoten kommunizieren. Im Gegensatz zum TCP-Protokoll, bei dem zwei Einweg-Verbindungen zwischen den Knoten erzeugt werden, sind es beim verbindungslosen AMUDP-Protokoll zwei Duplex-Verbindungen. Jeder Prozess auf einem Knoten verwendet für die insgesamt N Prozesse aller Knoten N-1 Puffer für die Speicherung von ausgehenden Requests und weitere N-1 Puffer für die Speicherung von ausgehenden Replys. Es werden mindestens 2*N-1 Puffer für eingehende Nachrichten verwendet. Puffer für ausgehende Nachrichten verwenden eine Beschreibungstabelle, welche den Status von Requests und Replys festhält. Die Tabelle beinhaltet Informationen über

- Benutzung des Puffers
- Sequenz Nummer
- Zeitüberschreitung.

Es hat sich herausgestellt, daß das binäre Wechseln des Sequenznummer von GASNet zwischen 0 und 1 in den Datenpaketen nicht ausreichend ist. Daher wurden Sequenznummern zwischen 0 und 15 eingeführt. Diese stellen unter der SX-Platform sicher, daß keine alten Nachrichten wiederverwendet werden, die nach zweimaligen Wechsel der Sequenznummer teilweise im Puffer waren.

D.2 Infiniband

D.2.1 Aufbau

Infiniband ist eine Technologie zur seriellen Übertragung von Daten zwischen Knoten (Punkt-zu-Punkt-Verbindung) mit hoher Geschwindigkeit und niedriger Latenz (unter 2 Mikrosekunden) [54]. Der bidirektionale serielle Bus leistet pro Kanal eine maximale theoretische Übertragungsrate von 2,5 Gbit/s in beide Richtungen. Es können transparent mehrere Kanäle gebündelt werden. Üblich sind zur Zeit die Bündelung von vier bzw. zwölf Kanälen. Als Übertragungsmedium dient Kupfer, welches eine Distanz von bis zu 15 m überbrücken kann. Bei Entfernungen von mehreren Kilometern werden fibreoptische Übertragungsmedien gewählt. Ein typischer Software Stack für Infiniband ist der OpenFabrics Stack [55]. Es existieren mehrere unterschiedliche IB APIs. Das hier

verwendete API ist das Verbs IB API [56]. Die vier Elemente, die man einem Infiniband-Netzwerk findet sind:

- Host Channel Adapter (HCA)
- Target Channel Adapter (TCA)
- IB Switch
- IB Router.

Der HCA ist eine IB-Netzwerkkarte, die auf Rechenknoten verwendet wird. TCAs werden für IO-Systeme wie RAID-Systeme verwendet. TCAs und HCAs können mehrere Ports besitzen, die mit unterschiedlichen IB Switches verbunden sein können. Jedem Port ist eine Local ID und ein Global Unique Identifier (GID) zugeordnet. Der IB Switch verbindet alle Channel Adapter (TCAs und HCAs) eines IB Subnets miteinander. Die Adressierung erfolgt mit der Local ID. Der Router ist im Prinzip auch ein Switch, der jedoch Daten zwischen Subnets versendet. Die Adressierung erfolgt hier über die GID.

D.2.2 Übertragung

Um zu verstehen wie in IB-Daten übertragen werden, wird zunächst auf die verschiedenen Arten eingegangen, wie mit IB Hardware kommuniziert werden kann. Die niedrigste Schicht zur Kommunikation mit IB Hardware ist das IB Verbs Interface [56]. Es beinhaltet einen Treiber auf Kernel-Ebene und eine optionale User Mode-Bibliothek, die direkt mit dem IB Hardware kommunizieren kann. Die Programmierschnittstellen von IB [57] können grob in zwei Kategorien unterteilt werden:

- Portable Interface
- Channel Access Interface.

Das Channel Interface wird benutzt, um mit dem Treiber des Channel-Adapters direkt zu kommunizieren. Ein Portable Interface ist eine übergeordnete Schicht zum Channel Interface, das die Interna des Channel Interfaces vor der User-Applikation verbirgt. Die GASNet-Implementierung legt eine solche Schicht auf das Channel Interface, die die Benutzung mit Active Messages verwendet. Da diese Schicht sehr dünn ist und direkt das Channel Interface benutzt wird, wird im folgenden genauer auf dessen Funktion eingegangen.
Damit ein Prozess mit einem anderen über Infiniband kommunizieren kann, muss zuerst eine Work Queue bestehend aus einem Queue Pair (QP) erzeugt werden. Das Queue Pair enthält unter anderem einen Sende- und einen Empfangspuffer. Damit ein Prozess eine Operation ausführen kann, muss ein Work Queue-Element in die Work Queue plaziert werden. Von da an wird das Work Queue-Element vom Infiniband Channel-Adapter aufgegriffen und ausgeführt. Die Work Queue bildet also das Kommunikationselement

zwischen Applikation und dem Infiniband Channel-Adapter. Durch diesen Mechanismus wird das Betriebssystem entlastet. Jeder Prozess kann eine oder mehrere Queue Pairs zur Kommunikation mit anderen Prozessen generieren. Anstatt sich um eine einzelne Queue einer Netzwerkkarte zu bewerben, wie es in einem typischen Betriebssystem üblich ist, hat jedes Queue Pair einen assoziierten Kontext. Da das Protokoll und die Strukturen sehr klar definiert sind, können Queue Pairs in Hardware implementiert werden. Dadurch wird die CPU entlastet. Nachdem ein Queue Pair-Element ordnungsgemäß abgearbeitet wurde, wird ein Completion Queue-Element erzeugt und in die Completion Queue gestellt, die von der Applikation abgefragt werden kann. Der Vorteil der Benachrichtigung der Applikation mithilfe eines Completion Queue-Elements innerhalb des Completion Queues ist die Reduzierung von Interrupts, die anderenfalls erzeugt werden würden.
Die Liste von Operationen, die von Infiniband auf der Transportschicht unterstützt werden ist für Send Queues wie folgt:

- Send/Receive
- RDMA-Write
- RDMA-Read
- RDMA Atomics.

Send/Receive unterstützt typische send/receive-Operationen, bei der ein Prozess eine Nachricht abschickt und ein anderer Prozess die Nachricht empfängt. Ein Unterschied zwischen der Implementation von send/receive-Operationen unter Infiniband und traditionellen Netzwerkprotokollen ist die Bindung der send/receive-Operationen an Queue Pairs. Die RDMA Write-Operation erlaubt es einem Prozess, direkt in den Speicherpuffer eines entfernten Prozesses zu schreiben. Der entfernte Prozess muss vorher geeignete Zugriffsrechte an den lokalen Prozess vergeben haben und auch Speicher für einen entfernten Zugriff registriert haben. Dasselbe gilt analog für RDMA Read-Operationen. RDMA Atomics-Operationen bezeichnet zwei Operationen, die denselben Zweck verfolgen, jedoch unterschiedlich voneinander arbeiten:

- Compare and Swap
- Fetch and Add.

Die Compare and Swap-Operation erlaubt es einem Prozess, einen Speicherort zu lesen und den Inhalt mit einem spezifischen Wert zu vergleichen. Wenn der Inhalt gleich einem spezifizierten Wert ist, dann wird ein neuer Wert in den Inhalt geschrieben. Die Fetch and Add-Operation liest einen Wert und gibt diesen an den aufrufenden Prozess zurück. Dann wird ein spezifizierter Wert zu dem gelesenen Wert addiert und an den ursprünglichen Speicherort zurückgeschrieben. Der Speicherort kann in der Zwischenzeit nicht verändert werden.
Für Receive Queues ist die einzige Operation Post Receive Buffer. Diese Operation identifiziert einen Puffer, in den ein Prozess Daten durch Send, RDMA-Read oder RDMA Write-Operationen senden oder empfangen kann.

Wenn eine Queue Pair erzeugt wird, kann der aufrufende Prozess zwischen fünf verschiedenen Infiniband Transport Service-Typen wählen:

- Reliable Connection (RC)
- Unreliable Connection (UC)
- Reliable Datagram (RD)
- Unreliable Datagram (UD)
- Raw Datagram.

„Reliability" bedeutet hier hauptsächlich die Zuverlässigkeit der Datenlieferung, d.h. Daten werden bei nicht ordnungsgemäßem Erhalt beim Empfänger neu gesendet. Dies gilt für verbindungsorientierte Connections und verbindungslose Datagramme. Bei Raw Datagrams wird die QPN (Queue Pair Number) nicht mitgeschickt. Die QPN ist Teil der Link Layer-Adresse und bezeichnet ein bestimmtes QP (Queue Pair) auf der Gegenseite. Das GASNet Extented API für Speicher put/get-Operationen benutzt im Infiniband Conduit Infiniband RDMA-Operationen. Es gibt zwei Conduits in GASNet, die Infiniband unterstützen:

- vapi Conduit
- ibv Conduit.

Das vapi Conduit wurde zuerst in GASNet unterstützt und basiert auf dem Mellanox-VAPI. Das ibv Conduit wurde erst später hinzugefügt und unterstützt den OFED (Open Fabrics Enterprise Distribution) Software Stack. OFED bietet eine breite Unterstützung von IB Hardware und ist als Open-Source verfügbar. Der Aufbau von OFED (im Schichtmodell von oben nach unten) ist wie folgt:

- Benutzerschicht
  - Infiniband Services
- Kernelschicht
  - Oberer Layer
    - IPoIB (IP over Infiniband)
    - SDP (Socket Direct Protocol)
    - SRP (SCSI RDMA Protocol)
    - iSER (Internet SCSI Extensions for RDMA)
  - MAD (Management Datagram) Clients und Services
  - Infiniband Verbs
  - Hardware-Treiber.

GAS-Sprachen unterscheiden sich von traditionellen Message-Passing-Schnittstellen, indem sie eine Möglichkeit bieten, Shared Memory-Programmierung durchzuführen mit gleichzeitiger Kontrolle über das Daten-Layout von Message-Passing-Systemen. Während dieses Modell gut auf eng-gekoppelten SMP Systemen funktioniert, existieren Design-Restriktionen auf herkömmlichen Netzwerken wie Infiniband. Dies gilt speziell für Remote DMA-Operationen. Hier ist es erforderlich, dass die entfernten Speicherbereiche als pinned markiert sind, da eine entfernte DMA-Speicheroperation auf eine Speicherseite, die eventuell ausgelagert wird, wenig sinnvoll ist.
Eine Strategie, die verwendet wird, um Remote DMA-Operationen in pinning-basierten Netzwerken zu unterstützen, nennt sich „Firehose“ [58]. Diese Strategie wird auch im Infiniband Conduit von GASNet verwendet. Der verwendete Firehose-Algorithmus versucht im allgemeinen Fall, einseitige zero-copy-Kommunikation zur Verfügung zu stellen. Um GAS-Sprachen effizient zu implementieren, ist dies notwendig, da darunterliegende Frameworks wie GASNet ja einseitige Operationen wie put/get verwenden. Die grundlegende Idee von Firehose ist es, die Performance von einem Pin-All-Ansatz (alle pinnable Speicherbereiche werden gepinnt) im allgemeinen Fall zu nehmen und auf einen Rendezvous-basierten Ansatz im speziellen Fall zurückzugreifen. In beiden Fällen versucht der Algorithmus, die Kosten von Synchronisierung und Pinning über mehrere entfernte Speicherbereiche zu amortisieren. Dies wird erreicht durch Steigerung der Performance über Rendezvous zur Vermeidung von Handshake-Nachrichten und durch die Vermeidung der Kosten für Repinning von kürzlich besuchten Speicherseiten.

E Speicherhandhabung

E.1 Memory Mapping

Unter Memory Mapping versteht man die Einblendung von Speicher. Der Grund warum man Speicher einblendet kann sowohl die Vereinfachung des Zugriffs auf eine Ressource, wie auch die Beschleunigung des Zugriffs auf eine Ressource sein. Prinzipiell unterscheidet man für die Einblendung folgende Typen:

- Teil einer Ressource, über die mit Dateideskriptor zugegriffen wird, wird in Speicherbereich eingeblendet
- physikalischer Speicher in virtuellen Speicher
- IO-basiert: Speicher von externem Gerät in Arbeitsspeicher.

Die Kommunikation mit den tatsächlichen Resourcen erfolgt dann über Speicherzugriffe. Da im Rahmen des IO-Forwarding Zugriffe auf Dateien ein zentraler Punkt sind, wird nachfolgend auf die Speichereinblendung von Resourcen über die mit Dateideskriptor zugegriffen wird, näher eingegangen.
Beim Schreiben in den eingeblendeten Speicherbereich einer Datei geschieht ein indirekter Schreibzugriff auf die betreffende Datei. Eine Datei oder allgemeiner eine Ressource, die über einen Dateideskriptor angesprochen wird, ist über eine Byte-für-Byte-Beziehung in den virtuellen Speicher eingeblendet. Diese Einblendung erlaubt es Applikationen diesen Speicherbereich zu behandeln, als wäre der Bereich gewöhnlicher Arbeitsspeicher.
Der primäre Vorteil der Einblendung von Dateien in den Speicher ist erhöhter IO-Durchsatz, besonders bei kleinen Dateien. Der Zugriff auf Dateien über Speichereinblendung ist aus zwei Gründen schneller als der direkte Zugriff über Lese- und Schreiboperationen:

1. Ein System Call ist um mehrere Größenordnungen langsamer als eine Änderungen von lokalem Speicher.
2. In den meisten Betriebssystemen ist die Speichereinblendung der Datei tatsächlich der Datei-Cache des Kernels, was bedeutet, dass keine Kopien im User Space angefertigt werden müssen, was wiederum unnötiges Kopieren von Speicher vermeidet.

Bestimmte Operationen auf den eingeblendeten Speicherbereich auf Applikationsebene verhalten sich performanter als direkte Dateioperationen. Applikationen können die Daten einer Datei direkt an der betreffenden Stelle ohne Seek-Operationen verändern.

Auch ein Umschreiben der kompletten editierten Teile einer Datei in einen temporären Bereich wird somit verhindert. Da die Speichereinblendung einer Datei intern in Speicherseiten gehandhabt wird, ist der Zugriff auf die Festplatte nur nötig, wenn auf eine neue Speicherseite zugegriffen wird. Der Zugriff auf die Festplatte kann somit mehr Daten innerhalb einer einzelnen Operation am Stück transportieren. Um die Speichereinblendung von großen Dateien effizient zu handhaben, können auch Teile einer Datei eingeblendet werden. Es handelt sich hier zumeist um die Teile einer Datei, die verändert werden. Dieses Prinzip ähnelt dem Demand Paging-Schema, das für Applikationen benutzt wird.
Wenn auf große Dateien zugegriffen wird, wird nicht alles vom Kernel in den Cache-Speicher aufgenommen. Dies bedeutet eine erhöhte Anzahl von Seitenfehlern, die sehr teuer bezüglich Zeitaufwand sind. Es ist somit möglich, dass herkömmliche Schreib- und Leseoperationen effizienter sind als die Methode der Speichereinblendung. Der Zugriff auf den physikalischen Speicher vom Gerät aus gesehen sollte über eine IOMMU erfolgen, da sich damit der Zugriff auf Dateien, die größer als der physikalisch adressierbare Speicherbereich bzw. verfügbare Speicherbereich sind, vereinfacht.
Da Dateioperationen wie open, read und write im Falle von IO-Forwarding System Calls benutzen, die an einen IO-Server über das Netzwerk weitergeleitet werden, macht es daher keinen Sinn Speichereinblendung für den IO-Forwarding Client zu verwenden. Der IO-Server könnte durchaus die Technik der Speichereinblendung für Dateien verwenden. Die gegenwärtige Implementierung benutzt jedoch auch System Calls auf der IO-Server Seite, d.h. es wird bei IO-Client und IO-Server aufgrund der Verwendung von sehr großen Dateien keine Speichereinblendung während des IO-Forwardings benutzt.

E.2 Pinning von Speicher

Alle Speichersegmente, die nicht auf sekundären Speicher ausgelagert werden sollen, können als nicht auslagerbar markiert werden. Man spricht dann von „pinned down memory“. Datenpuffer, die außerhalb der CPU sind und DMA-Speicher verwenden, sind typische Kandidaten für nicht auslagerbaren Speicher. Wäre der Speicher auslagerbar, würde der Prozess der DMA-Übertragung unnötig durch eventuelles Paging kompliziert und verzögert. Pinning von Speicher ist ein wichtiger Punkt bei der RDMA-Übertragung in einem Infiniband-Netzwerk.

E.3 Alignment

Die meisten CPUs erfordern es, daß Objekte und Variablen an bestimmten Offsets im Speicher liegen. Zum Beispiel 32-bit Prozessoren erfordern es, dass 4-byte Integer-Variablen an Speicheradressen liegen, die ganzzahlig durch vier teilbar sind. Intel-Prozessoren erlauben es, auch andere Adressen zu verwenden, was sich jedoch äußerst nachteilig auf die Performance auswirkt. Da der Speicher in Speicherseiten („Memory Pages“) unterteilt ist, sind diese Seitenstartadressen auch Vielfache von der Speicherseitengröße.